A Wide Spectrum Type System for Transformation Theory

Ph.D. Thesis

Matthias Ladkau

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Software Technology Research Laboratory

De Montfort University

2009

To Masha and my parents
for their love, support and encouragement
during this time of challenges.

Declaration

I declare that the work described in this thesis is original work undertaken by me between April 2006 and February 2009 for the degree of Doctor of Philosophy, at the Software Technology Research Laboratory (STRL), De Montfort University, United Kingdom. It is submitted for the degree of Doctor of Philosophy. Apart from this degree, no other academic degree or award was applied for based on this work.

Publications

M. Ladkau, A Revival for Legacy Systems - Successful Reengineering of Software Systems, in Proceedings of Informatiktage (GI, 2008), Volume S-6 of LNI, pp. 77-80 (2008).

M. P. Ward, H. Zedan, M. Ladkau, S. Natelberg, *Conditioned semantic slicing for abstraction; industrial experiment.* Software, Practice and Experience, Volume 38, Issue 12, pp. 1273-1304 (2008).

Acknowledgments

I would like to express my deepest gratitude to my supervisors Prof. Hussein Zedan and Dr. Martin Ward for their advice, experienced guidance and encouragement during the last years. I can truly say that this time was one of the most important and influential periods of my life. I would also like to thank my sponsor SML Ltd. for making all this possible. Without their supporting help the whole research project would not have been possible.

I am especially grateful to Maria Mogilnaya for her love, support and encouragement which changed my life. There are no words which can express how much I appreciate and value this. Further special thanks go to my parents for their invaluable constant support, their respect and understanding which gave me the courage to do this PhD. This thesis is dedicated for the ones I love with all my heart.

Finally, I would like to thank all of the colleagues in Software Technology Research Laboratory at De Montfort University for their valuable suggestions and discussions especially: Stefan Natelberg, Peer Bartels, Sascha Westendorf, Keno Buss, Feng Chen and Shaoyun Li. I am indebted to all of them.

A Wide Spectrum Type System for Transformation Theory

Abstract

One of the most difficult tasks a programmer can be confronted with is the migration of a legacy system. Usually, these systems are unstructured, poorly documented and contain complex program logic. The reason for this, in most cases, is an emphasis on raw performance rather than on clean and structured code as well as a long period of applying quick fixes and enhancements rather than doing a proper software reengineering process including a full redesign during major enhancements. Nowadays, the old programming paradigms are becoming an increasingly serious problem. It has been identified that 90% of the costs of a typical software system arise in the maintenance phase. Many companies are simply too afraid of changing their software infrastructure and prefer to continue with principles like "never touch a running system". These companies experience growing pressure to migrate their legacy systems onto newer platforms because the maintenance of such systems is expensive and dangerous as the risk of losing vital parts of sources code or its documentation increases drastically over time. The FermaT transformation system has shown the ability to automatically or semi-automatically restructure and abstract legacy code within a special intermediate language called WSL (Wide Spectrum Language). Unfortunately, the current transformation process only supports the migration of assembler as WSL lacks the ability to handle data types properly. The data structures in assembler are currently directly translated into C data types which involves many assumptional "hard coded" conversions. The absence of an adequate type system for WSL caused several flaws for the whole transformation process and limits its abilities significantly. The main aim of the presented research is to tackle these problems by investigating and formulating how a type system can contribute to a safe and reliable migration of legacy systems. The described research includes the definition of key aspects of type related problems in the FermaT migration process and how to solve them with a suitable type system approach. Since software migration often includes a change in programming language the type system for WSL has to be able to support various type system approaches including the representation of all relevant details to avoid assumptions. This is especially difficult as most programming languages are designed for a special purpose which means that their possible programming constructs and data

types differ significantly. This ranges from languages with simple type systems whose programs are prone to unintended side-effects, to languages with strict type systems which are constrained in their flexibility. It is important to include as many type related details as necessary to avoid making assumptions during language to language translation. The result of the investigation is a novel multi layered type system specifically designed to satisfy the needs of WSL for a sophisticated solution without imposing too many limitations on its abilities. The type system has an adjustable expressiveness, able to represent a wide spectrum of typing approaches ranging from weak typing which allows direct memory access and down casting, via very strict typing with a high diversity of data types to object oriented typing which supports encapsulation and data hiding. Looking at the majority of commercial relevant statically typed programming languages, two fundamental properties of type strictness and safety can be identified. A type system can be either weakly or strongly typed and may or may not allow unsafe features such as direct memory access. Each layer of the Wide Spectrum Type System has a different combination of these properties. The approach also includes special Type System Transformations which can be used to move a given WSL program among these layers. Other emphasised key features are explicit typing and scalability. The whole approach is based on a sound mathematical foundation which assures correctness and integrates seamlessly into the present mathematical definition of WSL. The type system is formally introduced to WSL by constructing an attribute grammar for the language. Type checking and type inference are used to annotate the Abstract Syntax Tree of a given WSL program with type derivations which can be used to reveal and indicate possible typing errors or to infer types if the program did not feature explicit type declarations in the first place. Notable in this approach is also the fact that object orientation is introduced to a procedural programming language without the introduction of new semantics. It is shown that object orientation can be introduced just by adjusting type checking rules and adding some syntactical notations. The approach was implemented and tested on two case studies. The thesis describes and discusses both cases in detail and shows how a migration which ignores type systems could accidentally introduce errors due to assumptions during translation. Both case studies use all important aspects of the approach, including type transformations and object identification. The thesis finalises by summarising the whole work, identifying limitations, presenting future perspectives and drawing conclusions.

Table of Contents

1 Introduction **1**

1.1 Motivation and Aim of Research 1

1.2 Research Method 4

1.3 Research Question 6

1.4 Research Hypotheses 6

1.5 Scope of Thesis 7

1.6 Original Contributions 8

1.7 Organisation 9

2 Background and Related Research **11**

2.1 Introduction 11

2.2 Software Engineering and the Software Crisis 12

2.3 Maintaining Legacy Applications 14

2.4 Software Evolution 16

2.5 Formal Methods and Software Evolution 18

2.6 History of FermaT 19

2.7 FermaT Program Transformation Theory 20

2.7.1 The WSL language 20

2.7.2 Mathematical Foundation of WSL 22

2.7.3 Semantics of a WSL Program 23

2.8 Proof Theoretic Refinement And Equality 25

2.9 Specification Statement 27

2.10 Flaws in the Current Migration Process 29

2.11 Other Migration Approaches 31

2.11.1 Migration of Assembler Code 31

2.11.2 Migration of High-Level Languages (HLL) . . . 32
2.12 History of Type Systems in Programming Languages . . . 33
2.12.1 Lambda Calculus . . . 33
2.12.2 Type Systems In Imperative Programming Languages . . . 35
2.12.3 Type Systems In Object Oriented Programming Languages . . . 39
2.12.4 Type Systems In Functional Programming Languages . . . 41
2.13 Concepts of Type Systems . . . 43
2.14 Characteristics of Type Systems for Programming Languages . . . 44
2.14.1 Type Checking Strategies . . . 44
2.14.2 Definition of Equality . . . 45
2.14.3 Safety of A Type System . . . 46
2.14.4 Accuracy of A Type System . . . 46
2.14.5 Scalability of Type Systems . . . 47
2.15 Type Inference . . . 47
2.16 Summary . . . 48

3 Preliminaries **50**
3.1 Introduction . . . 50
3.2 Composition of A Programming Language With A Type System . . . 51
3.3 Syntax of a Programming Language . . . 52
3.4 Abstract Syntax Tree . . . 54
3.5 Typing of a Programming Language . . . 55
3.6 Semantics of a Programming Language . . . 56
3.6.1 Operational Semantics . . . 56
3.6.2 Denotational Semantics . . . 57
3.6.3 Axiomatic Semantics . . . 57
3.6.4 Other Variants of Formal Semantics . . . 57
3.7 Attribute Grammars . . . 58
3.8 Example of a Language Definition . . . 61
3.8.1 Syntactical Definition: . . . 61
3.8.2 Semantic Definition (Operational Semantics): . . . 62
3.8.3 Type System Definition: . . . 63

3.9 Summary 64

4 Wide Spectrum Type System 65

4.1 Introduction 65

4.2 Approach of the Wide Spectrum Type System 65

4.2.1 Layered Approach 67

4.2.2 Emphasising Explicit Typing 69

4.2.3 Awareness of Varying Precision and Storage Size of Variables 70

4.2.4 Scalability 71

4.3 Introduction of the Wide Spectrum Type System to WSL 73

4.4 Mathematical Foundation 73

4.4.1 Attribute Grammar for WSL 74

4.4.2 Typing Judgments 74

4.4.3 Typing Rules 74

4.4.4 Correctness of Algorithms 75

4.4.5 Example 75

4.5 Type Checking and Type Inference 77

4.6 Summary 81

5 Anatomy and Realisation of the Wide Spectrum Type System 83

5.1 Introduction 83

5.2 General Definitions 84

5.3 Data Types of the Wide Spectrum Type System 85

5.3.1 VOID 85

5.3.2 POINTER 86

5.3.3 LIST 87

5.3.4 SET 88

5.3.5 SCALAR 88

5.3.6 STRING 88

5.3.7 INTEGER 89

5.3.8 BOOLEAN 89

5.3.9 HASH_TABLE 89

5.3.10 STRUCT . . . 90
5.3.11 REAL . . . 91
5.3.12 FIXED . . . 91
5.3.13 COMPLEX . . . 91
5.3.14 OSTRUCT . . . 92
5.4 Directives of the Wide Spectrum Type System . . . 95
5.5 Hierarchy of Types . . . 96
5.6 Dynamic Typed WSL . . . 97
5.7 Simple Typed WSL . . . 99
5.8 Weak Unsafe Typed WSL . . . 100
5.9 Strong Unsafe Typed WSL . . . 101
5.10 Strong Safe Typed WSL . . . 102
5.11 Object Oriented Typed Layer . . . 103
5.12 Summary . . . 104

6 Derivation, Verification and Transformation 105
6.1 Introduction . . . 105
6.2 Type Checking . . . 106
6.3 Type Inference . . . 109
6.4 Object Identification in Procedural Based Programs . . . 110
6.5 Type System Transformations . . . 112
6.5.1 Type Transformations For Procedural Typed Layers . . . 113
6.5.2 Type Transformations For Object Oriented Typed Layers . . . 120
6.6 Summary . . . 128

7 Tool Support 130
7.1 Introduction . . . 130
7.2 Development History . . . 131
7.3 FermaT Maintenance Environment . . . 132
7.3.1 Using the FermaT Transformation Engine . . . 133
7.3.2 Connecting the FME to the FermaT Engine . . . 135
7.3.3 Extracting the Transformation Catalogue . . . 137

7.3.4 Extracting and Generating Data Structures . . . 137
7.3.5 Interface of the FermaT Maintenance Environment . . . 144
7.3.6 Visual Representation . . . 145
7.3.7 Implementation . . . 147
7.4 FermaT Type System Editor . . . 153
7.4.1 Variable List . . . 154
7.4.2 Extracting and Generating Data Structures . . . 155
7.4.3 Interface of the Type System Editor . . . 155
7.4.4 Implementation . . . 156
7.4.5 Integration into the FermaT Transformation Process . . . 161
7.5 Summary and Conclusion . . . 164

8 Case Studies 165
8.1 Introduction . . . 165
8.2 Case Study 1 . . . 166
8.3 Case Study 2 . . . 172
8.4 Summary . . . 178

9 Conclusion and Future Research 180
9.1 Summary of the Thesis . . . 180
9.2 Limitations . . . 182
9.3 Conclusions and Future Directions . . . 183

Bibliography 185

A Source Code 193
A.1 Type Checking / Type Inferencing Algorithm . . . 194
A.2 Case Study 1 . . . 210
A.2.1 Source Code in FORTRAN 77 . . . 210
A.2.2 Source Code in Weak Unsafe Typed WSL . . . 217
A.2.3 Source Code in Object Oriented Typed WSL . . . 222
A.2.4 Source Code in Java . . . 228

B Data Structures for Code Analysis 234

B.1 Lexer Token List 235
B.2 Lexer Table 240
B.3 Tree Node List 242
B.4 Tree Node List Continuation (PrettyPrintTemplate Column) 249
B.5 JavaCC Definition for Untyped WSL 257

C FermaT Transformation Catalogues 287

C.1 Transformation Catalogue of fermat3 (Open Source Version) 288
C.2 Transformation Catalogue of fermat2 (Commercial Version) 294

D FermaT Maintenance Environment Tutorial 301

List of Figures

2.1 Development stages of a software systems in which vulnerabilities can be introduced 12
2.2 WSL Language Levels 21
2.3 Semantics of a WSL Program 24
2.4 Disproof of Arsac's transformation 27

3.1 Example Lexer Tokens 54
3.2 Example Parse Tree 55
3.3 Example Abstract Syntax Tree 55
3.4 Data Flow Between Inherited and Synthesised Attributes 59
3.5 Abstract Syntax Tree With Attributes 61
3.6 Example Type Derivation Tree 63

4.1 Layered Wide-Spectrum Type System 68
4.2 Abstract Syntax Tree Program 1 77
4.3 Abstract Syntax Tree Program 2 77
4.4 WSL Example Code 78
4.5 Type Check Example 80
4.6 Type Inference Example 80

5.1 Hierarchy of Data Types 96

6.1 Localise Item 121
6.2 Globalise Item 122
6.3 Insert Public Procedures 122
6.4 Localise Procedures 123
6.5 Insert Public Variables 124
6.6 Localise Variables 125

6.7 Merge Classes . . . 125
6.8 Partition Class . . . 126
6.9 Reduce Inter-Class Relations . . . 126
6.10 Create Composition . . . 127
6.11 Extract Super Class . . . 127
6.12 Declare Inheritance . . . 128

7.1 FermaT Maintainers Assistant . . . 131
7.2 FME UML Use Case Diagram . . . 132
7.3 FermaT Command Line . . . 134
7.4 Communication between the FME and the FermaT engine . . . 135
7.5 Example Abstract Syntax Tree . . . 142
7.6 Example Code with Connected Tree Nodes . . . 144
7.7 FermaT Maintenance Environment . . . 144
7.8 Example Call Graph (Procedures) . . . 145
7.9 Example Call Graph (Action System) . . . 146
7.10 Example Transformation History Graph . . . 146
7.11 UML Diagram of Framework and Logging Classes . . . 150
7.12 UML Diagram of configuration, GUI and Data Structure Classes . . . 151
7.13 UML Diagram of Sub-component Classes . . . 152
7.14 TSE UML Use Case Diagram . . . 153
7.15 Program and Variable List . . . 154
7.16 Type System Editor . . . 156
7.17 UML Diagram of Controlling, GUI, Algorithms and Data Structure Classes . . . 159
7.18 UML Diagram of Algorithm Classes (cont.) . . . 160
7.19 Current Migration Process to C . . . 162
7.20 Enhanced Migration Process to C . . . 163

8.1 Relationship of Modules . . . 166
8.2 Object Identification . . . 169
8.3 Object Identification . . . 170
8.4 UML Diagram of Case Study 1 . . . 171

8.5 Call Graph of Case Study 2 177
8.6 UML Diagram of Case Study 2 178

List of Tables

1.1 Data Type Lengths in C on Various Platforms 3

2.1 Level of Programming Languages 15
2.2 Weakest Precondition Example 25

4.1 Typing Rules for Example 79

5.1 Overview of Data Types 86

6.1 Excerpt of Typing Rule Table for Strong Safe Typed WSL 107
6.2 Initial Type System Transformation Bank 113
6.3 Example Insert Variable Declarations 114
6.4 Example Insert Type Annotations 114
6.5 Example Hex String to Number 115
6.6 Example Separate Multi-Typed Scalars 116
6.7 Example Insert Type Casts 116
6.8 Example Specify Scalar Variables 117
6.9 Example Declare Pointer variable 117
6.10 Rewrite SYMBOL Variables 118
6.11 Rewrite Pointer Variables for Arrays and Structures 119
6.12 Rewrite INTEGER to BOOLEAN Variables 119
6.13 Rewrite LIST Variables to SET Variables 120

7.1 Columns of the Lexer Token List 139
7.2 Columns of the Lexer Table 140
7.3 Columns of the Tree Node List 141
7.4 Pretty Print Commands 142

7.5 Excerpt from the Tree Node List 143
7.6 Example Processing of the Pretty Printer 143

8.1 Data Type Conversions of Case Study 2 173

Listings

1.1 Example of a Faculty Function in C . . . 2
1.2 Example of a Faculty Function in Fortran 77 . . . 2
2.1 Simple Code Example . . . 46
3.1 Example Code . . . 54
3.2 Example Programs . . . 62
4.1 WSL Example Code . . . 78
5.1 POINTER Example . . . 86
5.2 LIST Example . . . 87
5.3 HASH_TABLE Example . . . 89
5.4 STRUCT Example . . . 90
5.5 COMPLEX Example . . . 91
5.6 OSTRUCT Example (1) . . . 92
5.7 OSTRUCT Example (2) . . . 93
5.8 OSTRUCT Example (3) . . . 94
6.1 Pseudo Code for Type Checking Algorithm . . . 108
6.2 Pseudo Code for Object Identification Algorithm . . . 111
7.1 Start of FermaT Engine (Windows) . . . 136
7.2 Start of FermaT Engine (Linux) . . . 136
7.3 Communication Streams of the Pipe . . . 136
7.4 Migration Log . . . 161
8.1 Code Snipped from C Source . . . 167
8.2 Code Snipped from Corresponding WSL Source . . . 167
8.3 Code Snipped from C Source . . . 168
8.4 Code Snipped from Corresponding Wrongly Translated FORTRAN Source . . . 168
8.5 Code Snipped from WSL Source . . . 169

8.6 Code Snipped from WSL Source . . . 170
8.7 Code Snipped from C Source . . . 173
8.8 Corresponding Code Snipped from WSL Source . . . 173
8.9 Code Snipped from C Source . . . 174
8.10 Corresponding Code Snipped from WSL Source . . . 175
A.1 Type checking / Type Inferencing Algorithm in Java . . . 194
A.2 case_study.for . . . 210
A.3 analysis.for . . . 210
A.4 part_int.for . . . 212
A.5 lib.for . . . 214
A.6 case_study.wsl . . . 217
A.7 analysis.wsl . . . 217
A.8 part_int.wsl . . . 218
A.9 lib.wsl . . . 220
A.10 case_study.wsl . . . 222
A.11 analysis.wsl . . . 223
A.12 part_int.wsl . . . 224
A.13 lib.wsl . . . 225
A.14 global.java . . . 228
A.15 analysis.java . . . 229
A.16 analysis1.java . . . 229
A.17 analysis.java . . . 231
B.1 Parser Definition for Untyped WSL . . . 257

"We can't solve problems by using the same kind of thinking we used when we created them."

Albert Einstein

Chapter 1

Introduction

Objectives

- Motivate the use of type systems in current software migration projects.
- Formulate the research question.
- Outline the scope of the thesis.
- Highlight original contributions.
- Give a brief overview of the thesis organisation.

1.1 Motivation and Aim of Research

The constant innovation process of technology relegates an increasing number of existing software systems to old legacy systems. Many of today's legacy systems were implemented in low and medium level languages for various reasons: the most common probably being performance. Fortunately, with recent improvements in processor performance and compiler technology, raw performance is less of an issue than other limitations of low and medium level languages such as bad code comprehensibility due to high complexity or cryptic code statements and unreliability due to hidden side effects. Assembler and C are infamous examples for such languages [Moy92]. High level languages have emerged in recent years which focus on the construction of abstract models

with a high abstraction from the details of computer architecture. Key features like unambiguous syntax, well defined coding policies and encapsulation help to produce code with less side-effects and a clear comprehensible design. The Java language from Sun Microsystems is an example of this new mainstream [GM95]. Unfortunately, many companies do not take advantage of such technologies. Capers Jones, and Harry Sneed and Chris Verhoef give concrete numbers on the current situation of traditional corporations like banks, insurance companies, utility providers, retailers distribution, etc. Almost 70% of all business critical software runs on mainframes where 10% of the code is written in assembler (approximately 140-220 billion lines of code), 20% in C or C++ (approximately 180 billion lines of code) and 30% in COBOL (approximately 225 billion lines of code) [SV01, Jon98]. Companies who still rely on these legacy systems feel a growing pressure to migrate their legacy systems onto newer platforms as the maintenance of such systems is expensive and dangerous. However, despite the rising pressure many companies refrain from migrating their systems onto more recent platforms because of costs and/or the risk of instability a migration may cause. Unfortunately, this fear is not without reason as critical instability can be introduced during the migration process due to wrong code translations or by unidentified side effects which did not harm the system on the legacy platform. Consider for example the following C code which is a function to calculate the mathematical faculty of a number:

```
long facul(int f) {
  long i,ret;
  ret = 1;
  for (i=1;i<f+1;i++) {
      ret = ret * i;
  }
  return ret;
}
```

Listing 1.1: Example of a Faculty Function in C

Assuming that this function is part of a bigger system which is to be translated into Fortran during a migration project, the translated code might look as follows:

```
FUNCTION facul(f)
IMPLICIT NONE
INTEGER facul,f
INTEGER ret,i
ret = 1
```

```
i = 1
DO WHILE (i<f+1)
  ret = ret * i;
  i = i + 1
ENDDO
facul = ret;
END
```

Listing 1.2: Example of a Faculty Function in Fortran 77

The problem in this migration is that the translation is based on a common assumption in C regarding the length of data types:

```
sizeof (int) = sizeof (long)
```

The length of a data type determines its storage capacity i.e. the range of concrete numbers which can be stored. In this case the assumption is that the storage size of an integer variable is equivalent to the storage size of a long variable. However, as the following table[1] demonstrates, this is only true for certain platforms:

Length data type int	Length data type long	Platform Examples
16	32	PDP-11 Unix (later, 1977), Early MC6800
32	32	IBM 370, VAX Unix, Convex, some Microsoft operating systems
32	64	Most 64/32 Unix systems

Table 1.1: Data Type Lengths in C on Various Platforms

Unfortunately, this table is not the final truth as it is also possible, at least with some compilers, to alter the length of data types by using special command line switches. To make the situation even worse, the length is not the only problem which makes software migration such a difficult business. Another problem may arise with the encoding of data: Data can be stored in little endian or big endian format and numbers may be stored in binary, BCD (Binary Coded Decimal), Packed-Decimal or even proprietary formats (e.g. on older CRAY platforms). Migrating too quickly

[1] Data taken from John Mashey, The Long Road to 64 Bits, 2006 [Mas06].

and in a dilettantish way can be as expensive as staying too long on an old platform [Dug07]. Being aware of this situation, the research community has worked on several approaches to solve these problems since the early 90's. Meanwhile, Software Reengineering and Software Evolution have become well established research areas. Particularly program transformation technology has proved to be highly successful as a practical solution for industry. The approach described in this thesis extends the FermaT transformation system which is an industrial-strength approach utilising provably correct program transformations to restructure, abstract and simplify legacy systems by preserving or refining the semantics which guarantees that the transformed program logic is equivalent to the original program logic. FermaT uses an own intermediate language called Wide Spectrum Language (WSL) to represent and transform legacy code. Unfortunately, WSL does not define specific data types which, in turn, has created many problems in the past. The presented research aims to give insight into the nature of these problems and proposes a rigorous approach to solve them.

1.2 Research Method

The research area of this thesis belongs to software engineering which is usually a rigorous discipline aiming to enable successful production of high quality software with certain constrains such as time, costs, security and/or safety. As is also the case with the majority of computer science research the described research belongs to the field of constructive research where "constructive" refers to contributions to knowledge being developed as a new algorithm, method, model, framework or theory. As this investigation has been funded by industry and developed in an academic environment, the presented research in this thesis aims to be both: highly practical and academically rigorous. The thesis will make use of formal methods which can be defined as mathematically based languages, techniques, and tools for specifying and verifying systems. Formal methods can be used to increase the stability and reliability of a system by revealing inconsistencies, ambiguities, and incompleteness [CW96]. The presented research was realised as follows:

Step 1: Identification of the Problem, Research Question and Hypotheses.

At the outset of this scientific investigation was a problem which needed to be addressed. An understanding of the problem in its full scope was obtained by initial literature studies. Previous

research projects which had an impact on the identified problem were gathered and analysed. Most helpful for this were digital resources such as ACM Digital Library, CiteSeer, IEEE Xplore, and SpringerLink. Search engines such as Google were also used for discovering and for crosschecking the relevant information. Unfortunately, only a small number of papers and books had a significant impact on the type system related topics which were important for the presented research. Notable in this sense are the books: "Types and Programming Languages" by Benjamin C. Pierce [Pie02] as well as "The Art of Compiler Design: Theory and Practice" by Thomas Pittman and James Peters [PP92].

Step 2: Construction of an Abstract Solution Model and Implementation of the Approach.

A major challenge was the nature of WSL as an intermediate language, able to represent code constructs for various languages with different type system approaches. As the relevant problems and requirements became more and more clear, the initial idea of a layered type system which is able to adjust to different type system approaches was considered. A prototype software was then developed to demonstrate the applicability of the proposed approach and to make assessment from the academic- and engineering perspectives possible.

Step 3: Validation and Verification of Hypotheses.

The conducted hypotheses was verified by case studies and validated by means of defined criteria. The case studies demonstrated the feasibility of the approach and showed that it is possible to obtain reliable results in reasonable time. The case studies were carefully selected to be representative and to show the potential application space of the targeted application domain.

Step 4: Deriving Conclusions.

Conclusions were drawn from the experiences of the evaluation. New research questions were raised to motivate further research in this area.

1.3 Research Question

To give a direction for the investigation, a research question was raised and formulated. The research was driven by a number of problems which are inherent in current software migration technology. This thesis tries to answer the following overall research question:

How can a type system contribute to a safe and reliable migration of legacy systems in current software migration projects?

To answer this question in its full spectrum and to address certain issues in detail, a number of sub questions were formulated:

1. What are the key aspects of type related problems in the FermaT software migration and how can they be solved?
2. How can a mathematical foundation of the approach be formulated to make it reliable?
3. How can the type system be enforced?
   - 3.1. How can the type information be represented?
   - 3.2. How can the process of type checking by done accurately?
   - 3.3. How can the process of type checking be efficient and scalable?
4. Which type system properties should be included into the new type system approach?
   - 4.1. What are the properties of type systems in current legacy systems?
   - 4.2. What are the properties of type systems in potential migration target languages?
   - 4.3. How is it possible to cope with different levels of type strictness?
5. How can tool support be provided for the proposed research?

1.4 Research Hypotheses

1. A type system can solve many migration related problems and contribute significantly to the accuracy of a migration result. Especially, problems related to the separation of code and

data, as found in the current FermaT migration approach, which may not recognise faults and even introduce new errors to a migrated software system.

2. No single type system approach is suitable for the Wide Spectrum Language of FermaT as the type systems of potential migration target languages and potential migration source languages are very different in terms of "typefulness"[2] and type strictness.

3. The approach should focus on procedural languages as migration source languages as most of today's legacy systems are written in procedural oriented code.

4. A semi-automatic approach which allows manual intervention should be favored for the implementation because not enough information can be gathered from source code alone. Especially, when migrating into object oriented languages.

5. Despite a sound mathematical foundation it is not possible to migrate a legacy system into a new language with a 100% accuracy. Every implementation of migration software has to find a good "mixture" of accuracy, practicability and efficiency.

1.5 Scope of Thesis

This thesis is not about advances in type theory. It is rather a structured approach to develop a sophisticated type system for softare migration. The type system itself is actually a composition of many different existing type system approaches. The thesis concentrates on the motivation, description and realisation of this multi layered type system. Its scope includes:

1. Analysing the current FermaT migration process and identify certain flaws which can be tackled by a sophisticated type system.

2. Definition of key features and mathematical foundation.

3. Definition of a type checking strategy which finds a balance between reliability and effectiveness.

4. Definition of all type system layers with their features and data types.

[2] See the paper of Cardelli [Car91] for an explanation of this term.

5. Definition of type system transformations to move a given program among the type system layers.

6. Implementation of the type system using FermaT's Wide Spectrum Language (WSL) as code representation language and the Type System Editor as supporting tool to perform type checks and apply type system transformations.

7. Description of two case studies which evaluate the approach.

1.6 Original Contributions

This thesis includes the following original contributions:

1. The most significant contribution is the proposal of a Wide Spectrum Type System. A type system for a single programming language with adjustable expressiveness and strictness. The combination of the Wide Spectrum Language with the Wide Spectrum Type System is a major contribution for current software migration technology.

2. An approach for migration of procedural oriented code into object oriented code has been developed. The definition of special OSTRUCTs with procedure pointers allows a step by step migration from procedural code into object oriented code.

3. Type system transformations have been defined which are able to move a given program among the layers of the wide spectrum type system i.e. to change the type strictness and "typefulness" of the program.

4. For each layer of the Wide Spectrum Type System a slightly different WSL language definition had to be developed. These definitions usually includes new data types and sometimes also new program statements (e.g. TYPEDEF, DEFINE, ...).

5. A set of supporting tools has been developed which demonstrates the applicability, scalability and reliability of the approach.

1.7 Organisation

The thesis is organised in chapters, sections and subsections. Each of these entities can be unambiguously identified by a specific number. Figures and tables have unique two part numbers whereby the first number describes their chapter while the second number is continuous counted within the chapter. A list of all figures, listings and tables can be found after the table of contents in the beginning of this document. The document structure is as follows:

Introduction: The current introductory chapter which includes the motivation and aim of the presented research, the research method used, the initial research questions, the conducted research hypotheses, the document scope, the original contributions and this outline.

Background and Related Research: A literature review covering all related background information which influenced the proposed approach. This includes software engineering and software evolution, formal methods, the FermaT transformation system and its theory as well as history, concepts, characteristics and implementation details of type systems.

Foundation of the Wide Spectrum Type System: Defines key features, the mathematical foundation and efficient enforcement strategies are described and certain aspects and implications of these definitions are discussed.

Preliminaries: Gives an overview about essential concepts which are needed to understand the presented approach.

Anatomy and Realisation of the Wide Spectrum Type System: This chapter describes every type system layer with all its data types and features in detail.

Algorithms for Derivation, Verification and Transformation: All related algorithms which are used to introduce, alter or check the type system are described and discussed in this chapter.

Tool Support and Evaluation: The implementing tools are presented, described and evaluated. This chapter shows also how the proposed approach could be integrated into the current FermaT migration approach.

Case Studies: Two case studies are evaluated in this chapter to demonstrate the practical applicability of the research.

Conclusion and Future Research: This chapter summarises the whole thesis, evaluates the success of the research and draws conclusions. It also describes certain limitations of the approach and states suggestions for future research.

"We always strain at the limits of our ability to comprehend the artifacts we construct - and that's true for software and for skyscrapers."

James Arthur Gosling

Chapter 2

Background and Related Research

Objectives

- Present and summarise related work of the thesis.
- Introduce basic concepts related to software reengineering and software evolution.
- Evaluate existing approaches of type systems in common programming languages.
- Discuss general properties of type systems for programming languages.
- Review the FermaT system as example of current reengineering and migration technology.
- Review related reengineering and migration approaches.

2.1 Introduction

This chapter reviews and summarises related work for the FermaT transformation system and the Wide Spectrum Type System. It presents some critical aspects of current software engineering and explains why so many of today's software products are unstable. It will furthermore introduce and discuss the FermaT transformation theory and will elaborate some weaknesses in its current implementation due to the absence of a sophisticated type system in WSL. The chapter will furthermore describe the long history of type systems in programming languages, their concepts and characteristics and how they can be used to make program development more reliable.

2.2 Software Engineering and the Software Crisis

In times where industry has realised the vision of ubiquitous and pervasive computing [Wei91] the demand for specialised software is continuously rising. Software has become a highly successful commercial product. Unfortunately too much of recently produced software is late, over budget and shows far too often unexpected behavior. This is the result of low-price / time-to-market pressure and the fact that the resource allocation for software development projects are often done by "non-technical" managers who have little or no knowledge about the technical details [Bro75]. Software very quickly becomes complex and the development effort is very hard to estimate. The calculated development time of software is almost always too short despite the knowledge that many defects are only revealed after extensive testing. The result of this can be seen in many of today's unstable and insufficient software products. Software is in fact vulnerable in every step of its realisation process, from specification to concrete implementation. With each realisation step

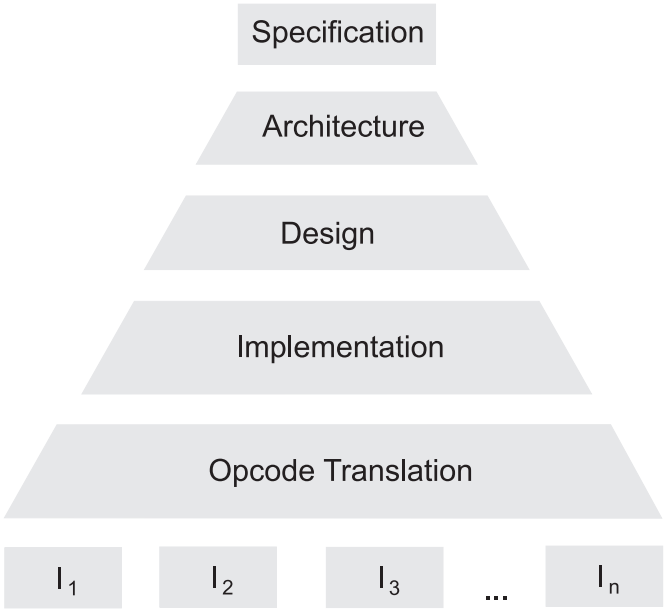

Figure 2.1: Development stages of a software systems in which vulnerabilities can be introduced

vulnerabilities might be introduced which may cause the software:

- to malfunction or show unintended behaviour;
- to produce incorrect or corrupted results; or

- to be eligible for attacks against its integrity due to malicious intents.

Errors or misconstruction introduced in the upper layers are most serious as each succeeding layer is highly dependent on its predecessors. As early as 1972 Edsger W. Dijkstra [Dij72] expressed the issues of software engineering precisely in two sentences:

> "*To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products.*"
>
> *Edsger W. Dijkstra, The humble programmer, 1972*

Research projects have been discussing this problem and since the 1960s it has become a well-established research area known as *software engineering*. Many approaches have been researched to "tame" the so called *software crisis* [NR69] but as Brooks stated in his essay [Bro87] there is "no silver bullet", which means none of the approaches will be able to solve the whole problem.

> "*There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.*"
>
> *Frederick P. Brooks, Jr., No Silver Bullet, 1987*

In his essay he identifies four properties of irreducible essence of modern software systems and gives reasons why he thinks that the software crisis cannot be solved and no "silver bullet" will ever be found.

Complexity This is an inevitably property of all large software systems. In mathematics and physics great progress was made by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This is possible because the complexities ignored in the models were not the essential properties

of the phenomenon. Abstracting away the complexity of a software entity will often also abstract away its essence which would make it useless.

Conformity Another significant amount of complexity comes into a software when it is constrained to conform to interfaces of a human institution or other systems.

Changeability It is much easier to change a software product (e.g. via a patch or update) than an automobile or some other complex object from the real world. After an automobile is sold it is unlikely that its features will be changed a lot. Vendors will introduce new features when a new series of these automobiles are produced. Software on the other hand is just a collection of Bits in memory. It is common practice to produce a rather unstable software product and introduce it to the market as soon as possible. Then, after the first money has been generated, the development concentrates on releasing patches to fix the most serious bugs - many products of Microsoft and their Service Packs are an infamous example of this. Comparing this to the real world, no one would ever buy a car with deficient brakes or only 3 tires.

Invisibility When constructing a complex physical object the designers and constructors have blue prints and floor plans which provide an accurate overview. For complex software there is no unifying geometric representation of its structure. The only way to describe software in detail with all relevant information is by several distinct but coherent and interacting graphs.

The statement “no silver bullet” does not mean that there is no solution at all for the problems, but there will not be a general solution. Every approach has and will have advantages and drawbacks. Therefore it depends on the system and its environment if an approach is appropriate or not.

2.3 Maintaining Legacy Applications

Reviewing today’s software systems which are used in industry it becomes clear that most of the huge software systems have a long development history including numerous modifications and extensions. Much knowledge and money has been put into these systems over the years to make them useable and sufficiently stable. Considering this it is no surprise that most companies are simply too afraid about changing their software infrastructure over night. However, depending on

the age of the software the risk of losing vital parts of source code or its documentation increases drastically. This can be due to damaged backups, bad version management, lost compilers or malfunction of old hardware which cannot be replaced. Furthermore, as the system has changed over time there is a growing danger that malicious code with negative side effects may have been introduced. Fatally, the introduced code by itself could be clean and free of errors whereby the side-effects arise when the code interacts with other components of the system. A large number of these errors occur because of unclean interfaces and usage of data which implies "defaults" (e.g. floating point numbers are not simply truncated but rounded when they are converted to integers). Most interfaces are poorly or not at all documented, which forces programmers to make dangerous assumptions about possible input, output and error values. Implementing a function point in a "low-level" language requires nearly two to three times as many lines of code (which may include faults) as in a higher level language (e.g. Assembly (Basic) to C or C to Java), and costs nearly three times as much [Jon96]. Notable is that the vast amount of costs (over 90%) does not arises in the development stage, but in the maintenance stage when adjustments and new functionalities are added to the system [Bro75]. The following table is taken from Programming Languages, Table Release 8.2, March 1996 [Jon96] by Capers Jones, Chairman, Software Productivity Research, Inc.: A "low-level" language is much harder and more expensive to maintain, than a high level language.

Language	Language Level	Average Source Statements per Function Point
Assembly (Basic)	1.00	320
Assembly (Macro)	1.50	213
C	2.50	128
COBOL	3.00	107
FORTRAN 77	3.00	107
C++	6.00	53
JAVA	6.00	53

Table 2.1: Level of Programming Languages

It can be difficult to carry out extensive enhancements to a legacy system if not impossible if the developers of the system have left the company. A recent example of such a situation happened in the US state of California. Since the state is about $15bn in debt, the government was forced to take action by cutting the salaries of California's 200,000 state employees and by firing 10,000 employees additionally. Unfortunately, the reconfiguration of the government's aging COBOL-based payroll system would have taken six months. State controller John Chiang refused to issue reduced pay checks because it was simply not possible to change the system quickly enough. The

irony of this example is that the only ones who could have made the changes - part time retired COBOL programmers - were among the 10,000 fired employees [Man08]. Such examples are, unfortunately, rather the rule than the exception. A recent survey (Computerworld survey of IT managers) revealed that 62% of the surveyed organisations still use Cobol in their organisation. By way of comparison 61% use Java, 26% use C, 23% use C# and 7% use Fortran. For most of the surveyed companies, which use Cobol, the software was an internally developed business application and 58% of all companies which use Cobol still develop new business applications in this language [Mit06]. In a recent assessment of Gartner Inc. regarding the age of software languages and tools [Dug07] it was stated that:

> "*Most organizations are overwhelmed with volunteers to try out the "hot" new technologies. Few find a similar level of excitement when it comes to legacy languages and tools. However, the legacy application portfolio continues to run the business, and a revolutionary replacement of languages and tools is simply not feasible in most organizations. ... Taking a longer view, a legacy language such as COBOL has at least another decade of useful life, with few practical threats for small and midsize businesses on the horizon.*"
>
> *Jim Duggan (Gartner Inc.), Assessing the Age of Software Languages and Tools, 2007*

2.4 Software Evolution

A very critical requirement for a software to be usable for commercial purposes is that the system must be continuously adapted to current business processes. This continuous process of software reengineering is today known as software evolution [YW03]. Lehman formalised over a period of twenty years eight *Laws of Software Evolution* [Leh96]:

Continuing Change A program that is used must be continually adapted else it becomes progressively less satisfactory.

Increasing Complexity As a program is evolved its complexity increases unless work is done to maintain or reduce it.

Self Regulation The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.

Conservation of Organisational Stability The average effective global activity rate on an evolving system is invariant over the product life time.

Conservation of Familiarity During the active life of an evolving program, the content of successive releases is statistically invariant.

Continuing Growth Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

Declining Quality Programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

Feedback System Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

Industrially used large-scale software system usually have a long life-cycle. During this time the software evolution process will complete many cycles. Unfortunately, some changes might also have bad side effects such as reduced performance or decreased reliability [Par94]. Four types of software changes can be identified [Swa76, LBSB80]:

Perfective Changes to improve or enhance the product (e.g. adding new user requirements, enhance performance, usability, etc.)

Corrective Changes which fix defects in the system.

Adaptive Changes to integrate the system into changing environments (e.g. new operating systems, language definitions, database management systems, etc.).

Preventive Changes to enable or improve future maintainability and reliability of a system.

Because software evolution is a critical though essential process for every business company, researchers are constantly investigating ways to make it as safe and reliable as possible. In recent years formal methods have become more and more interesting for this purpose, as they provide ways to “prove” correctness.

2.5 Formal Methods and Software Evolution

In the recent past formal methods in the form of mathematically based languages and transformation techniques have proven to be suitable for specifying and verifying software systems. This can be extremely helpful for the whole software evolution process. Baumann et. al. [BFK+94] for example even stated that reverse engineering methods must be based on a sound (formal) foundation to avoid the extraction of wrong information during the reverse engineering process. Wrong information can introduce new errors or side-effects into the program being maintained. To achieve this he suggests formal denotational semantics as a foundation. Another application, for formal methods within the software evolution process, is to increase the comprehension by revealing inconsistencies or ambiguities [CW96]. An approach for Software Evolution which uses formal methods should consist of the following essential components [YW03]:

1. The semantic model has to be a sound mathematical logical structure where all terms, formulas and rules have a precise meaning.
2. A specification language is used to describe the intended behavior of the system.
3. The verification system and the refinement calculi are rules that allow verification of properties and refinement of specifications.
4. Defined development guidelines show how a method should be used.
5. Supporting tools supply the proof-of-concept for a proposed method.

Two main advantages of using formal methods can be identified. First, formal methods provide a rigorous and precise description of the described system. This can greatly increase the quality of the new system. Secondly due to the soundness of a formal method approach the reengineering process can be automated provably correct in many parts. This may decrease the costs for reengineering dramatically [YW03]. One fundamental achievement in this area has concentrated on program transformation theory and is realised within an industry-strength toolset called FermaT [WB95, War99, War04] released under the General Public License (GPL) and currently available[1] for Microsoft Windows and Linux/Unix.

[1] http://www.cse.dmu.ac.uk/∼mward/fermat.html

2.6 History of FermaT

The roots for the transformation theory of FermaT go back to 1989 [War89]. In the early 1990s a prototype transformation system was developed at Durham University. The implementation language of the so called "Maintainer's Assistant" [WCM89] was LISP. Although it already including a large number of transformations it was very much an "academic prototype". The main aim of which was to test and evaluate the ideas. More precisely, little attention was paid to the time and space efficiency of the implementation. Nevertheless the tool proved to be highly successful and was able to reengineer moderately sized assembler modules into equivalent high-level language programs. The Maintainer's Assistant already utilised a special intermediate language called WSL (Wide Spectrum Language) on which all the transformations operated. At that time, programs were represented as LISP structures and the transformations were written in LISP. The next version of the tool was called GREET (Generic Reverse Engineering Tools). It was decided to extend the WSL language to add domain-specific constructs, creating a language for writing program transformations. The extensions included an abstract data type for representing programs as tree structures, constructs for pattern matching, pattern filling and iterating over components of a program structure. The extension was called $\mathcal{META}$WSL [WZ05] and all transformations from the Maintainer's Assistant, typically with enhancements, were rewritten in $\mathcal{META}$WSL and translated to LISP via the Concerto case tool builder. Additional many new transformation were added to the transformation catalogue. For the latest version (called FermaT) the underlying runtime language was changed from common LISP to Scheme. All remaining parts of the system were reimplemented in $\mathcal{META}$WSL and a $\mathcal{META}$WSL to Scheme translator was implemented (also in $\mathcal{META}$WSL) which was then used to bootstrap the whole system to a Scheme implementation in a few weeks work. The FermaT system is now implemented almost entirely in $\mathcal{META}$WSL. To enhance the usability of the FermaT tool a new graphical interface called "FermaT Maintainer's Environment" was developed in 2006. This graphical interface is written entirely in the object-oriented language Java and can be used to learn about FermaT and to test the ideas of research projects which relate to transformation theory.

2.7 FermaT Program Transformation Theory

The FermaT transformation system uses formal proven program transformations, which preserve or refine the semantics of a program while changing its form. These transformations are applied to restructure and simplify legacy systems, currently with emphasis on assembler systems, and to extract higher level representations (specifications). By using a suitable sequence of transformations, the extracted representation is guaranteed to be equivalent to the original code logic. Equipped with advanced code slicing techniques the FermaT transformation system is additionally very useful for analysing tasks [War04, WZ06]. In contrast to simple line by line language migration technologies, FermaTs semantics preserving code transformations enable the original application to be automatically cleaned-up, simplified and restructured. Once migrated, these systems are substantially easier to maintain and to evolve. Furthermore, it is ensured that only functional code is migrated to the new language. The process used by the FermaT transformation system consists of three basic steps [War04]:

1. Translation of legacy code to WSL;
2. Translate and restructure data declarations;
3. Apply semantics-preserving WSL to WSL transformations;
   (a) For migration: translate the high-level WSL to the target language.
   (b) For analysis: apply slicing or abstraction operations to the WSL to raise the abstraction level even further.

2.7.1 The WSL language

The core of the FermaT transformation system is the WSL language. It is based on a small kernel language which itself is based on infinitary first order logic. WSL contains all the operations needed for a programming and specification language including Dijkstra's guarded commands [Dij76] and a specification statement which is able to represent any WSL program [War04]. The intention is to form a language which acts as an intermediate language when processing a legacy system. WSL was designed for reengineering tasks and covers:

- Simple, regular and formally defined semantics
- Simple, clear and unambiguous syntax
- A wide range of transformations with simple, "mechanically checkable" correctness conditions
- The ability to express low-level programs and high-level abstract specifications

The heart of the WSL language is a very small and mathematically tractable kernel language. This language already supports all necessary operations needed for a programming and specification language. In the context of this tiny kernel language it is relatively easy to prove the correctness of a transformation, but the language is not very expressive for programming. For that reason the language is extended into an expressive programming language by defining new constructs in terms of the kernel. In his paper [War04] Ward describes the extension in a series of layers with each layer building on the previous language level. The "Wide Spectrum" in "Wide Spectrum Language"

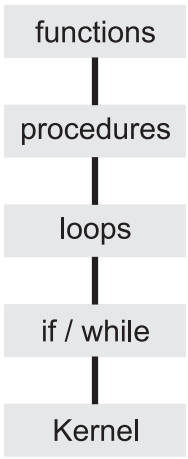

Figure 2.2: WSL Language Levels

refers to the range of operations, from "low level" things such as assignments, IF statements and GOTOs to high level operations such as specification statements. The language is also "wide" because it suits all stages of reverse engineering [YW03]. Besides having all the usual programming structures and commands, WSL also contains commands, functions and routines for operating on programs written in WSL. As mentioned before, this reflexive extension was named $\mathcal{META}$WSL. This gives the opportunity for specifying, writing, analysing, rewriting, and simplifying programs within the same language.

2.7.2 Mathematical Foundation of WSL

The Kernel language is based on infinitary first order logic [War04]. Infinitary logic, originally developed by Carol Karp [Kar74], is an extension of ordinary first order logic which allows conjunction and disjunction over (countably) infinite lists of formula, but quantification over finite lists of variables. The kernel is defined by just four primitive statements and three compound statements. Let **P** and **Q** be any infinitary logical formula and **x** and **y** be any finite lists of variables. The primitive statements are:

1. **Assertion:** $\{\mathbf{P}\}$ is an assertion statement which acts as a partial **skip** statement. If the formula **P** is true then the statement terminates immediately without changing any variables, otherwise it aborts (abnormal termination and non-termination are treated as equivalent i.e. a program which aborts is equivalent to one which never terminates);

2. **Guard:** $[\mathbf{Q}]$ is a guard statement. It always terminates, and enforces **Q** to be true at this point in the program *without changing the values of any variables*. It has the effect of restricting previous non-determinism to those cases which will cause **Q** to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including **Q**);

3. **Add variables:** $\mathbf{ADD}(\mathbf{x})$ ensures that the variables in **x** are in the state space (by adding them if necessary) and assigns arbitrary values to the variables in **x**. The arbitrary values may be restricted to particular values by a subsequent guard;

4. **Remove variables:** $\mathbf{REMOVE}(\mathbf{y})$ ensures that the variables in **y** are *not* present in the state space (by removing them if necessary).

The compound statements are:

1. **Sequence:** $(\mathbf{S}_1; \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;

2. **Non-deterministic choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution, the choice being made non-deterministically;

3. **Recursion:** $(\mu X.\mathbf{S}_1)$ where X is a *statement variable* (a symbol taken from a suitable set of symbols). The statement $\mathbf{S}_1$ may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is $\mathbf{S}_1$.

When thinking in terms of an ordinary programming language some of these constructs, particularly the guard statement, seem to be unfamiliar, while other constructs such as assignments and conditional statements are missing. The answer to that is that assignments and conditionals can be constructed out of these more fundamental constructs. For example:

WSL Construct	WSL Code	Representation in Kernel Language
Assignment	`x := 1`	$\mathbf{ADD}(\mathbf{x})$; $\{\mathbf{x} = 1\}$
if-then-else	`if B then S1 else S2`	$([\mathbf{B}]; S1) \sqcap ([\neg\mathbf{B}]; S2)$

In fact the guard statement by itself cannot be implemented in any programming language. For example, the guard statement [**false**] is guaranteed to terminate in a state in which **false** is true. In the semantic model this is easy to achieve: the semantic function for [**false**] has an *empty* set of final states for each proper initial state. As a result, [**false**] is a valid refinement for *any* program. Morgan [Mor94] calls this construct "miracle". Such considerations have led to the Kernel language constructs being described as "the Quarks of Programming": mysterious entities which cannot be observed in isolation, but which combine to form what were previously thought of as the fundamental particles. Further details on the definition of WSL are given in the article "Pigs from Sausages" by Martin Ward [War04].

2.7.3 Semantics of a WSL Program

The mathematical model for WSL defines the semantics of a program as a function from states to sets of states. A *state* is simply a function which gives a value from a given set H to each of the variables in a given set V of variables. The set V is called the *state space*. For each initial state s, the function f returns the set of states $f(s)$ which contains all the possible final states of the program when it is started in state s. The function f is called a *state transformation* which represents a collection of symbols structured according to the syntactic rules of infinitary first order logic, and the definition of the WSL kernel language. A special state $\perp$ indicates non-termination

or an error condition. A *state predicate* is a set of proper states - states other than $\perp$. If $\perp$ is in the

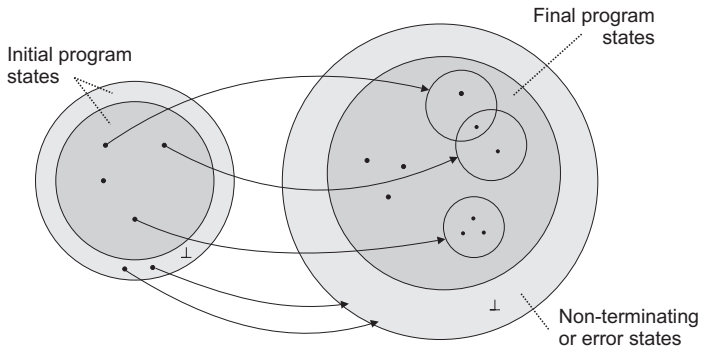

Figure 2.3: Semantics of a WSL Program

set of final states, then the program might not terminate for that initial state. If two programs are both potentially non-terminating on a particular initial state, they are consider to be equivalent on that state. A program which might not terminate is no more useful than a program which never terminates. It is not interesting despite whatever else it might do. The semantic function is defined to be as such that whenever $\perp$ is in the set of final states, then $f(s)$ must include every other state. This restriction also simplifies the definition of semantic equivalence and refinement. If two programs have the same semantic function then they are said to be *equivalent*. For further details see "Pigs from Sausages" and "Analysing and Abstracting Legacy Assembler Code via Conditioned Semantic Slicing" by Martin Ward [War04, WZ06]. If for example $=$ is defined as the equality relation then the interpretation of `x = y` on the value set $H = \{0,1\}$ is the *state predicate* $\{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$. Suppose the state $s_{ij} = \{x \mapsto i, y \mapsto j\}$. The state transformation function of the assertion statement $\{x = y\}$ is:

$$\{s_{00} \mapsto \{s_{00}\}, s_{01} \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}, s_{10} \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}, s_{11} \mapsto \{s_{11}\}, \perp \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}\}$$

The state transformation function of the guard statement $[x = y]$ is:

$$\{s_{00} \mapsto \{s_{00}\}, s_{01} \mapsto \{\}, s_{10} \mapsto \{\}, s_{11} \mapsto \{s_{11}\}, \perp \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}\}$$

The state transformation function of the add statement $\mathbf{ADD}(\mathbf{x})$ is:

$$\{s_{00} \mapsto \{s_{00}, s_{10}\}, s_{01} \mapsto \{s_{01}, s_{11}\}, s_{10} \mapsto \{s_{00}, s_{10}\}, s_{11} \mapsto \{s_{01}, s_{11}\}, \perp \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}\}$$

The state transformation function of the assignment `x := y` or $\mathbf{ADD}(\mathbf{x})$; $[x = y]$ is:

$$\{s_{00} \mapsto \{s_{00}\}, s_{01} \mapsto \{s_{11}\}, s_{10} \mapsto \{s_{00}\}, s_{11} \mapsto \{s_{11}\}, \bot \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \bot\}\}$$

2.8 Proof Theoretic Refinement And Equality

Two approaches can be used to define the *equality* or the *refinement* of a program [War04, WZ07].

Through the semantic function : Two programs are *equivalent* if their state transformation functions are identical. A program f_1 is *refined* by f_2, written $f_1 \leq f_2$, iff for every initial state s, $f_2(s) \subseteq f_1(s)$. The statement $\{\textbf{false}\}$ is *refined* by every other statement, while the statement $[\textbf{false}]$ *refines* every other statement.

Through the Weakest Precondition : Another way to define *refinement* and *equality* is the Weakest Precondition (WP) first introduced by Dijkstra [Dij76].

For a given program $\mathbf{P}$ and its state transformation function $f()$ and condition $\mathbf{R}$ on the final state space, the weakest precondition $\text{WP}(\mathbf{P}, \mathbf{R})$ is the weakest condition on the initial state such that if $\mathbf{P}$ is started in a state satisfying $\text{WP}(\mathbf{P}, \mathbf{R})$ then it is guaranteed to terminate in a state satisfying $\mathbf{R}$ (see table 2.2 for an example). All proper states satisfy *TRUE* and no program states satisfy *FALSE*. In his thesis [War89] Ward proves that $\mathbf{P}$ can be mapped to a state transformation f and $\mathbf{R}$ can be mapped to a state predicate e such that $\text{WP}(f, e)$ is the state predicate s which satisfies $f(s) \subseteq e$. The *refinement* relation using weakest precondition can be expressed as follows.

Program $\mathbf{P}$	Condition $\mathbf{R}$	$\text{WP}(\mathbf{P}, \mathbf{R})$
`x := 5`	$x > y$	$y < 5$
`a := 2 * b + 1`	$a = 13$	$2 * b + 1 = 13 \Rightarrow b = 6$
`if x = 1 then x := 5 else x := 6 fi`	$x > z$	$(x = 1 \wedge z < 5) \vee (x \neq 1 \wedge z < 6)$

Table 2.2: Weakest Precondition Example

A program F_1 is *refined* by F_2, written $F_1 \leq F_2$, iff:

$$\forall \mathbf{R}.(\text{WP}(F_1, \mathbf{R}) \subseteq \text{WP}(F_2, \mathbf{R}))$$

Instead of looking at all possible postcondition for a particular WP of statements, it is sufficient to check only two special postconditions being **true** and $x \neq x'$ where x is a list of all the variables

in the final state space and x' is a list of new variables not used elsewhere [War04]. For example to prove $F_1 \leq F_2$ which is the *refinement* of program F_1 by program F_2 under a set of assumptions Δ it is sufficient that the formula:

$$\mathrm{WP}(F_1, true) \Rightarrow \mathrm{WP}(F_2, true) \ \wedge \ \mathrm{WP}(F_1, x \neq x') \Rightarrow \mathrm{WP}(F_2, x \neq x')$$

can be proved (or deducted) from the set Δ of assumptions or *sentences* (formula with no free variables). This case is written:

$$\Delta \vdash F_1 \ \leq \ F_2$$

The proof that this definition of refinement is equivalent to the definition of refinement in terms of semantic functions is given in Wards thesis [War89]. If both $\Delta \vdash F_1 \ \leq \ F_2$ and $\Delta \vdash F_2 \ \leq \ F_1$ then F_1 and F_2 are *equivalent*, written: $\Delta \vdash F_1 \ \approx \ F_2$. A *transformation* is any operation which takes a statement F_1 and transforms it into an *equivalent* statement F_2 (where Δ is the set of *applicability conditions* for the transformation). An example of an "applicability condition" is a property of the function or relation symbols on which a particular transformation depends. For example, the statements $x := a \oplus b$ and $x := b \oplus a$ are *equivalent* when $\oplus$ is a commutative operation. The transformation can be written as:

$$\{\forall a, b.\, a \oplus b = b \oplus a\} \vdash x := a \oplus b \ \approx \ x := b \oplus a$$

To prove the *equality* or the *refinement* is of uttermost importance for a transformation theory as it is the only way to prove that a transformation is indeed correct. Experience showed that without such features the results may be faulty especially when the transformation of code is automated. An example of an approach with a weak formal foundation was described in an article by Jacques J. Arsac [Ars79] in 1979. In the article the following transformation was described (page 3):

$$\text{Let } f, g \in F$$

$$\text{do } f \text{ od} \equiv \text{do } g \text{ od} \Leftrightarrow \text{do } f \text{ od} \equiv \text{do } g;\ f \text{ od}$$

F here represents the set of sequences of statements (if-then-else, do ... od, assignments, etc.). Despite his claims that all presented transformations are correct by proof, it can be shown that at least some of them (e.g. the example transformation above) are not correct. Concerning the

example above, consider the statement:

$$f = \texttt{exit(1)}$$

$$g = \texttt{SKIP}$$

Because of the iff equivalence in the transformation definition the programs in figure 2.4 should be all equal. In fact, they are clearly not. While both programs on the right do exactly the same (terminate immediately), only the program in the upper left corner is semantically equivalent. The program in the lower left corner, however, will result in an infinite loop. This means that with this transformation it is possible to combine a non-terminating program with a terminating program and transform them into a terminating one which was clearly not intended by the author. The

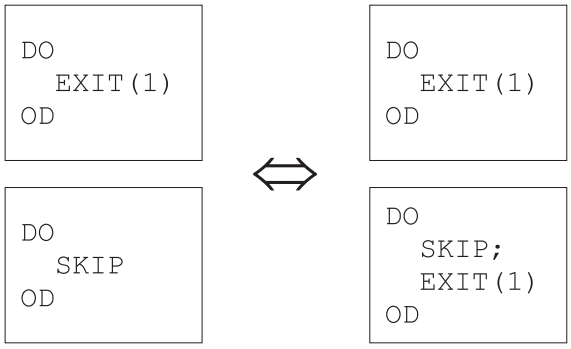

Figure 2.4: Disproof of Arsac's transformation

example shows that the notions of *equivalence* and *refinement* must be carefully defined, as they form in most cases very fundamental parts of any theory. Fortunately, in type theory, similar notions are easier to define: Equivalence can be related to the definition of type equality (see section 2.14.2), while refinement can be related to subtyping which is a rather common concept in many type systems (see section 2.13).

2.9 Specification Statement

As mentioned before the Wide Spectrum in the name "Wide Spectrum Language" refers to the range of operations which span from "low level" to "high level" [WZ06]. Abstract specifications are very useful for both forward and reverse engineering (see Bachman's Reengineering Cycle [Bac88]).

It is indispensable for a language designed for reengineering tasks to be able to represent those abstract specifications as part of the language. As a consequence the refinement of a specification into an executable program, or the reverse process of abstracting a specification from executable code, can both be carried out within a single language. This has motivated the definition of the *specification statement* for WSL. Informally, a specification describes what a program should do without saying explicitly how it should do that. A specification can be formalised as a list of variables (the variables whose values the program should change), a formula defining the relationship between the old and new values of the variables and any other required variables. A simple combination of kernel statements is used to construct the *specification statement*:

$$x := x'.Q$$

x is a sequence of variables (with old values) and x' the corresponding sequence of "primed variables" (with new values) while Q is any formula. The statement assigns new values to the variables in x so that the formula Q is true. If no new values for x can be found which satisfy Q then the statement aborts. For example, the specification statement for sorting the array A could be:

$$A := A'.(sorted(A') \wedge permutation(A', A))$$

In [War04] it is shown that *any* WSL program can be specified using a single specification statement. Within the kernel language the specification statement can be defined as:

$$x := x'.Q =_{DF} \{\exists x'.Q\};\ \mathbf{add}(x');\ [Q];\ \mathbf{add}(x);\ [x = x'];\ remove(x')$$

The initial assertion of the specification statement ensures that the *specification statement* is guaranteed to be *null*-free. This means that for every input state the set of output states is non-empty. A *null* program is a program for which the set of output states is empty for one or more initial states. For example the guard [false] can never be satisfied. Interestingly this program satisfies any specification since a specification must satisfy the given postcondition. A *null* program is therefore a correct refinement of any specification, but is also not implementable (since the physical machine must terminate in some state if it is guaranteed to terminate). To avoid the possible specification of unimplementable programs in the refinement process it is indispensably necessary to prove that a specification cannot specify a *null* program. This *null*-free property is the fundamental difference

to the specification statement of Morgan [MR87] which is directly derived from the weakest precondition. The Morgan specification statement is written $x : [Pre, Post]$ where *Pre* and *Post* are formulas of infinitary first order logic. The statement is guaranteed to terminate for all initial states which satisfy *Pre* and will terminate in a state which satisfies *Post*. Both formulas will thereby only assign to variables in the list x. In Ward's notation an equivalent statement is $\{Pre\}; add(x); [Post]$. The disadvantage of Morgan's specification statement is that the user is responsible for ensuring not to refine into an (unimplementable) *null* statement since the statement does not abort if the set of output states is empty. Furthermore, there is no guarantee that any program can be written as a single Morgan specification statement: the simple program `x := x + 1`, for example, cannot be written as a single Morgan specification statement.

2.10 Flaws in the Current Migration Process

At first glance the FermaT migration concept looks very reasonable and sound. However, the authors were slightly superficial in the second step of their migration process. By taking a closer look into the step "Translate and restructure data declarations", it turns out that the current process simply extracts the data structures and translates them into the target language. Although the data structures are stored in an (internal) intermediate data format during the migration process, they are never altered or checked throughout the whole migration process. Because WSL cannot represent any data types, the data structures are separated from the code at a very early stage of the migration process. The absence of a "sophisticated" type system which, according to Cardelli [Car91], is essential for any proper software evolution approach, has caused several flaws within the FermaT migration approach:

- Abstraction transformations on data structures are currently not possible. While WSL is a powerful programming language, able to represent source code at various levels of abstraction, data structures are only preserved and therefore kept at a low-level. A target language must be able to approximate the behavior and layout of the legacy data structures very closely. This becomes very difficult or even impossible if the type system of the target language is more strict or defines certain data types differently. Examples of these problems are the exclusion of unsafe data types in target languages (e.g. data types which allow direct memory

access in Java or C#) or a proprietary data format in source languages (e.g. floating point numbers on CRAY SV1 machines or all IBM System/360 based mainframes which use the IBM Floating Point Architecture).

- Hidden flaws like implicit type casting can cause a loss of data if data of current business rules are beyond the initial specifications. These errors are in most cases very serious because they may lead to wrong results only with certain input data. The current migration process is unable to trace these hidden errors as it migrates the data separately from the code.
- Pointers cannot be expressed at all. For these purposes the current implementation of FermaT uses the pseudo array a[] which models the memory of the computer. This can only be a temporary solution as it is not part of the language specification and makes WSL code ambiguous.
- Variables which are in a special data format (e.g. packed decimal format) cannot be distinguished adequately from normal variables.

So far the only working source language is assembler which can either be translated to C or COBOL. In the book [YW03] pp. 78-79 the authors claim:

> "*An obvious disadvantage of working in a separate language to the source language of the legacy system is that translators to and from WSL will have to be written. Fortunately, for the "old fashioned" languages typical of legacy systems, this is not much more difficult than writing a parser for the language, which in turn is a simple application of well-developed compiler technology for which there is a wide variety of tool support available.*"

In fact this cannot be true because of the above-mentioned drawbacks. The simple example presented in section 1.1 illustrates the possible flaws. Even if the C program is translated to FORTRAN through WSL, the mentioned problems remain as WSL does not support data types nor specification of lengths for data types. As shown in table 1.1 different computing platforms might define different lengths for data types. On these platforms the variables would lose some Bits of precision during operation. Fatally, the program would appear to work correctly at first and only later, depending on the input data, it might produce corrupted results. Because WSL

does not have an explicit type system, information like the length of data types would have to be added manually during the translation process. But even with this information the migration may fail as the program could still contain implicit type casts which may not have affected the original code but would cause the translated code to produce corrupted results. This would be the case, for example, if a type cast has to fill an `integer` variable with a `long` value which would be out of range only on the new platform where `long` and `int` are of different lengths. The current migration process comprises a great danger to introduce hidden errors as long as the real problem - the separation of code and data - is not solved. The presented research addresses this with the development of a special type system for WSL which takes all type related migration issues into account.

2.11 Other Migration Approaches

The comparison of the Wide Spectrum Type System to other approaches is very difficult as the FermaT migration approach itself is quite unique. The idea of an intermediate language which is able to represent programs on various levels as well as a fully automated migration approach which performs provably correct code abstraction transformations is not common for software migration tools. Also the idea of a type system with adjustable expressiveness can hardly be found in any publication since such a type system only makes sense for intermediate languages like WSL which need adaptation abilities to such an extent.

2.11.1 Migration of Assembler Code

Most automated migration approaches for assembler migration have taken the form of a "brute force" conversion simply mapping an assembler instruction directly to corresponding code in a high level language. The resulting code is often more complex and slower than the original code [WZH04]. The semi-automated approaches which require manual modification on the other hand usually use an own abstract representation of the program rather than a textual format. The Bogart tool of Feldman and Friedman's approach [FF95] for example uses data flow and control flow for manipulations. Bogart produced code which was between half and three quarters as large and more than twice as fast as a "brute force" conversion. However, prior application, extensive

manual modification of the assembler code was required. The consequence of this is that the migration becomes expensive (an experienced programmer can modify about 3600 lines of code per month) and error prone ("Manual preparation of the code has probably damaged the code's quality" [FF95]). Moreover, the translation result was hard to comprehend since it was assumed that "the resulting code was not required to be particularly readable"[FF95]. Taking all this into account even the authors concluded that: "extensive manual work is not only harmful for the resources it requires, it may also endanger the whole translation enterprise"[FF95].

2.11.2 Migration of High-Level Languages (HLL)

Despite, the 140-220 billion lines of assembler code which are still in use[SV01, Jon98], the majority of research towards industrial-strength software migration approaches focused on the migration of high-level languages. A considerable number of research projects have investigated the migration from procedural languages into object-oriented languages since the early 90s. The approaches ranged from wrapping the whole legacy system or some of its components in wrapper objects [Sne00] to highly complex processes which involve many steps. Most of them rely on the interaction with a maintainer to capture domain- and application-specific knowledge and use high level information such as dataflow analysis [Sne92, PZL98, GK95]. A rather uncommon approach was developed in 1998 which collected all procedures of a legacy system into one class (called god class) and then utilised "design-transformations" to split the system into many classes [PZL98]. Despite many years of research the success remains questionable. A recent survey among 59 Italian information technology companies showed that only 36% of the surveyed companies used special tool support for software migration[TDPR+08]. In contrast to assembler migration where supporting tools are desperately needed due to the masses of code which need to be handled, the migration of higher-level languages such as C, COBOL, etc. are often done without advanced tool support. In the majority of cases the surveyed companies used only some basic utilities provided by operating systems. Among the companies who used a tool, some used an ad-hoc proprietary tool developed by the company itself while others used commercial or free tools only for parts of the migration process. Only in 2 cases (5%) a specific migration toolset such as the Transformation Assistent[2] was used for the whole process. The majority of surveyed migration projects dealt with a programming

[2] http://www.relativity.com/pages/transformation-assistant.asp

language change. The targeted languages were mostly Java, C# and Visual Basic while the most common source language was COBOL. Interestingly, the main problem, which was identified by the authors, was the reengineering of data: "ensuring data consistency seems to be the main problem companies have during a migration" [TDPR+08].

2.12 History of Type Systems in Programming Languages

The proposed type system for WSL adapts and combines many popular type system approaches into one coherent model. Though it is not possible to combine all approaches into a single type system, it is possible to define a multi layered type system with distinct layers to model several approaches and defined transition transformations to move a program among these layers. The following sections describe the mathematical origin of type systems and some of the most popular interpretations for software technology. For the approach of the Wide Spectrum Type System, the approaches of imperative and object oriented programming languages where most influential.

2.12.1 Lambda Calculus

Originally type theories were a discipline of logic which subsequently became increasingly interesting for the programming languages in the area of computer science. The roots of type systems date back to 1940 when Alonzo Church presented his Theory of Simple Types [Chu40] as an extension to his lambda calculus. The lambda calculus was the result of an attempt by Alonzo Church and Stephen Kleene to formalise a generalised theory about functions and logic in the early 1930s [Kle35, Chu36]. The calculus forms a rewrite system consisting of a formal language together with some reduction and conversion rules. The advantage of the language was that the formal syntactical definition involved only 3 terms.

$$\text{T} ::= \text{[a-z]+} \mid (\text{T T}) \mid \lambda\ \text{a.T}$$

The terms are variables (with any name), application and abstraction or lambda. An abstraction accepts input and produces an output by rewriting all the occurrences of its input variable with

the input.

$$(\lambda x.x)t \rightarrow t$$

Functions with multiple arguments are modelled through cascading abstractions:

$$(\lambda x.(\lambda y.xy))$$

A simple example showing the basic abilities of the lambda calculus is a function to change the order of a triple:

$$(\lambda x.(\lambda y.(\lambda z.zyx)))abc \rightarrow cba$$

It is possible to model normal numbers and operations for the normal decimal system as well as for boolean algebra (binary numbers and operations like AND, OR, XOR, etc.). In 1936 Kleene discovered that this minimal model is expressive enough to formalise all properties of recursive functions. A year later Turing showed that the Turing computability is equivalent to the Lambda definability. This meant the lambda calculus was as powerful as every programming language. The approach can be seen as the first step towards encapsulation and reusable code as Peter Landin [Lan65] stated that most programming languages are rooted in the lambda calculus, which provides the basic mechanisms for procedural abstraction and procedure (subprogram) application. In fact most of the current functional programming languages like ML or Haskell primarily implement the lambda calculus with some extensions. One of the first was John McCarthy who used the lambda calculus in the late 50's for the core functions of the programming language LISP. The calculus was already very powerful but it lacked the strong normalisation property which means that every sequence of rewrites in a rewrite system eventually terminates to a term in normal form. The problem occurred when applying a term to itself. Consider, for example, the Lambda term:

$$(\lambda x.xxx)$$

Applying a normal variable to the function would rewrite to a normal form:

$$(\lambda x.xxx)a \rightarrow aaa$$

But applying the function to itself would never result in a normal form:

$$(\lambda x.xxx)(\lambda x.xxx) \rightarrow (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$$

Therefore the simply typed lambda calculus was introduced by Church in 1940 as an attempt to avoid these inconsistencies. With the simply typed lambda he introduced a strongly normalising version of the calculus. The given typing rules excluded the application of a function to itself. At first this was just an extension of the untyped lambda calculus but a more modern view considers the typed lambda calculi as the more fundamental theory and the untyped version only as a special case with only one type [Pie02]. Although strong normalisation is a very useful property for logicians it has the serious drawback that a language with this property is not Turing complete and therefore is not able to express all computable functions. All programs written in that particular language would always terminate. Due to the origins of the lambda calculus there is a gap between type theory and types in computer languages. In the logic area the calculi are still extended to form even more complex and powerful type systems while computer science tries to keep the type systems as simple and flexible as possible [PS94]. Nevertheless the introduction of types showed a possibility to define more clearly what the intention of a function is and how to prevent misusage.

2.12.2 Type Systems In Imperative Programming Languages

The first programming languages which appeared had of course more emphasis on performance than on a good structure. Computer programs were very machine dependent and much less complex than nowadays. Debugging could easily be done by inserting some PRINT statements and the few defined data types were directly supported by the underlying hardware. The main aim of a programming language was to ease the task of generating assembler code for formulas and to control the input/output devices. However, as soon as the programming languages introduced concepts like subroutines, global and local variables; the programs became increasingly complex and the need for types and type checking emerged. The most important reason for types was of course efficient resource allocation and fast object code but also to define a clean interface between two subsystems and to handle complex data structures (e.g. arrays, lists, tree structures, hash tables, and their combinations) in a safe way. The safety of types was accorded greater consideration than previously because the increased complexity of programs resulted in a growing danger of accidentally producing malicious or meaningless code. More and more different programming

languages appeared each with its own type system. The designer of a language had to make a decision between restriction, which reduces the applicability of the language, and freedom, which can result in a language producing unsafe code. Two extreme cases in the history were the languages FORTRAN and C.

The FORTRAN Language

The FORTRAN language first appeared in 1956 [Int56] for the IBM 704 computer and contained 32 statements. Most of them controlled the IO devices (e.g. READ DRUM, PUNCH). The FORTRAN I release had no procedural programming facilities but it already supported two important data types. The 2 data types could be allocated in 1, 2 or 3-dimensional arrays. A variable was either integer (called fixed point) or floating point. The declaration of a data type for a variable was done when the name was written. A fixed point variable began with I,J,K,L,M or N and floating point with any other character except these [3]. It was not possible to use both data types in one expression (e.g. $A = I * B$) although it was possible to convert between floating point and fixed point. Functions used in expressions were either built into FORTRAN or were accessible as a pre-written subroutine in 704 language on the FORTRAN master tape. The next version FORTRAN II emerged in the year 1958. This version included facilities for procedural programming. It was possible to put highly frequented code segments into a subroutine or, when a return value was required, into a function. Now it was also possible to write routines in FORTRAN itself. Furthermore, it introduced the COMMON block; the possibility to declare global variables. A FORTRAN III came up in the same year that allowed inline assembler but it was never released as a product. Over the next years FORTRAN II also supported the DOUBLE PRECISION and the COMPLEX data type. The great disadvantage of the FORTRAN languages was the machine-dependency. It was very difficult to port a program from one platform to another. Therefore in 1961 a decision was made to completely rewrite the FORTRAN compiler to improve speed and to remove the machine dependent features from the FORTRAN language. FORTRAN IV appeared in 1962 and now included the LOGICAL data type. As the popularity of FORTRAN was constantly increasing the American Standards Association (now known as ANSI) formed a committee to develop an “American Standard Fortran” which became the first industry-standard version of FORTRAN

[3] These conventions were still present in later FORTRAN versions. A FORTRAN program has to start with `implicit none` to avoid this implicit declaration.

known as FORTRAN 66. From that time FORTRAN was standardised by the ANSI. The standard was largely based on the FORTRAN IV. During the next years compiler vendors introduced their own extensions to "Standard Fortran" prompting the ANSI to revise the FORTRAN standard. In April 1978 a new standard was released known as FORTRAN 77. Along with other changes, the language now provided improved support for structured programming (e.g. block IF statements with ELSE and ELSE IF) and the CHARACTER data type for processing character-based data. Historically FORTRAN 77 is probably the most important dialect of FORTRAN because it was standard for about fifteen years. In 1992 the successor of FORTRAN 77, known as FORTRAN 90, was finally released. This version was a major step towards object-orientation although the real object orientation support came about in the FORTRAN 2003 release. FORTRAN 90 was now supporting modules to group procedures and data together, an improved argument-passing mechanism allowing interfaces to be checked at compile time and abstract/derived data types. Five years later a minor revision of the standard was undertaken to fix some outstanding issues from the Fortran 90 standard. The standard was then called FORTRAN 95. The most recently used standard is the FORTRAN 2003 standard which has now finally introduced object oriented programming support. Along with other modern features like type extension and inheritance, polymorphism, dynamic type allocation, type-bound procedures, allocatable components and deferred type parameters. FORTRAN is a current programming language with probably the longest history [All81, Bac98]. The emphasis since the very beginning has been to abstract from the machine towards mathematics.

> "*The FORTRAN language - closely resembling the ordinary language of mathematics, ... FORTRAN therefore in effect transforms the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself.*"
>
> *The FORTRAN I Manual, 1956*

It is an example of a well defined programming language which has fewer critical side-effects than C, for example. The property of type safety was introduced very early and for this reason the language has been able to "survive" over such a long period of time.

The C Language

Another important language in the history of programming languages is the C language. In terms of type safety it can be seen as the inferior opposite of the well defined FORTRAN language. The C language is a general-purpose computer programming language developed between 1969 and 1973 by Dennis Ritchie at Bell Telephone Laboratories for the Unix operating system. It is the most commonly used programming language for writing system software like drivers or kernel modules for operating systems. C is a successor of the B language which was developed by Ken Thompson in 1970. Although many important ideas of C stem from the BCPL language developed by Martin Richards. B is a stripped down version of the BCPL programming language. Dennis Ritchie developed C to create a new language which inherited Thompson's taste for concise syntax. The key to its success was a powerful mix of high-level functionality and detailed low-level features required to program an operating system. The main characteristic of the C language is the simple and minimal core language while any extra functionality such as mathematical functions and file handling is provided by library routines. Another important feature of C were fundamental data types like characters, integers and floating point numbers of several sizes but also compositional data types for higher order data types. Unlike BCPL and B which were 'typeless' languages. Consecutively most components of the Unix System were rewritten in C, culminating in the kernel itself in 1973. In 1978, Dennis Ritchie and Brian Kernighan published the first edition of The C Programming Language which served for many years as the informal specification of the language [KR78]. The described version of C is commonly referred to as "K&R" C. During the 1980s, it was adopted for use with the IBM PC, and its popularity began to increase significantly. At the same time, Bjarne Stroustrup and others at Bell Labs began work on adding object-oriented programming language constructs to C, resulting in 1985 in a language called C++ [Sto85]. Although C++ has become a widely spread programming language it has many drawbacks due to its roots in the C language [Joy92]. In 1983, the American National Standards Institute (ANSI) formed a committee to define a standard specification of C. The first official standard was completed in 1989 and ratified as "Programming Language C". This version of the language is often referred to as ANSI C, or sometimes C89 (to distinguish it from C99). The book by Ritchie and Kernighan has been revised and is now available as a second edition according to the new standard [KR88]. In 1990, the ANSI C standard (with a few minor modifications) was adopted by the International

Organization for Standardization (ISO) as ISO/IEC 9899:1990. This version is sometimes called C90. Therefore, the terms 'C89' and 'C90' refer to essentially the same language. One of the aims of the ANSI C standardisation process was to produce a superset of K&R C (the first published standard), incorporating many of the unofficial features which had been subsequently introduced. However, the standards committee also included several new features, such as function prototypes (borrowed from the C++ programming language), and a more capable preprocessor. The syntax for parameter declarations was also changed to reflect the C++ style. C89 is supported by current C compilers, and most C code being written nowadays is based on it. It appears that C has many disadvantages in terms of safety and portability. The simplicity of the language forces programmers to define all complex data structures (sets, hash tables, lists, trees, graphs etc.) in terms of pointers which can easily cause buffer overflows and memory leaks due to the absence of an automated garbage collection and array bounds checking. Another disadvantage are the many defaults and unofficial features which have been integrated into the language over the past years and which make so many programs platform and compiler dependent. In early versions of C for example, only functions that returned a non-integer value needed to be declared if used before the function definition; a function used without any previous declaration was assumed to return an integer. Many defaults have no obvious logical reason but they are there and a programmer needs to be aware of them. It is most likely that many software quality flaws stem from unknown defaults [Joy92] which would be reported as errors if the language had been defined in a more strict and safe manner.

2.12.3 Type Systems In Object Oriented Programming Languages

In 1990 the "object oriented programming" (OOP) emerged. A very foundational book proposing object oriented programming was published in 1988 by Bertrand Meyer [Mey97] who originally designed the now ISO-standardised OOP language Eiffel. The roots of this paradigm go back to the 1960's when the programming language Simula was created [Hol94] and when the nascent field of *software engineering* had begun to discuss the idea of the software crisis [NR69]. The OOP approach addressed the problem of maintaining software quality in a new way by strongly emphasising modularity and encapsulation along with other paradigms, particularly, inheritance and polymorphism. It uses "objects" to model real-world objects in software. The communication

with other objects is provided through a defined interface. These techniques are in fact there to constrain the programmer to access a certain software construct only through its interface. The advantage of this is that another programmer can change the software construct without introducing side-effects to other software components by simply providing the same interface as the old implementation. Although every constraint will more or less hinder the software development process, the interface constraint has far more benefits than disadvantages. In fact, it is most likely that this constraint is the reason for the great success of the OOP paradigm. A very current and popular language based on the OOP paradigm is the Java language.

The Java Language

The Java technology was originally created as a programming tool in a small, closed-door project from SUN called "The Green Project" initiated by Patrick Naughton, Mike Sheridan, and James Gosling in 1991. In summer 1992 the team emerged with a working demo. It was an interactive, handheld home-entertainment device controller called *7 (Star Seven). The Java language itself was created by James Gosling specifically for *7 [Byo98]. After failing to find a market for *7, the technology was developed further towards web applications. At that time the Internet was still difficult to use. And HTML was only able to present static content. With the Java technology, however, the web pages could be extended with dynamic features. It was now possible to move "behavior" in the form of applets along with the content. The new technology was demonstrated with a Mosaic based Web-Browser called "WebRunner" which later evolved into the "HotJava"-Browser. The first public release of Java and the HotJava web browser was on May 23, 1995 at the SunWorld conference. The initial Java Development Kit (JDK) was released on January 23, 1996. Since then the technology has matured and the current Java language features a huge API including over three thousand classes. Despite the dramatic enhancements over time, the primary goal of the Java language remains unchanged. To provide portable and platform independent code and to learn from the mistakes made in C and C++.

> "*Java must enable the development of secure, high performance, and highly robust applications on multiple platforms in heterogeneous, distributed networks. ... Java is designed for creating highly reliable software. It provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide program-*

mers towards reliable programming habits. The memory management model-no pointers or pointer arithmetic-eliminates entire classes of programming errors that bedevil C and C++ programmers. You can develop Java language code with confidence that the system will find many errors quickly and that major problems wasn't lay dormant until after your production code has shipped."

J. Gosling and H. McGilton, The Java Language Environment[GM95], 1995

The language favors a safe and strong type system which prohibits explicit pointer arithmetics. The vast number of available Java applications present proof that this is indeed not necessary for most applications. Especially this constraint, in fact, involves many more benefits than disadvantages.

2.12.4 Type Systems In Functional Programming Languages

The functional programming paradigm is directly derived from the previouly mentioned lambda calculus. It conceives computation as the evaluation of mathematical functions by avoiding state or mutable data. In contrast to imperative programming which emphasises the changes of states, the functional programming paradigm emphasises the application of functions [Hud89]. Often the languages feature multi-paradigms (e.g. LISP features functional, imperative and object oriented paradigms) but all functional programming languages can be categorised into pure and impure. Whereas pure functional programming languages (e.g. Haskell) have no side effects, allowing to rigorously reason about their behavior, a pure language, for example, can substitute $y = f(x) * f(x)$ into $z = f(x)$; $y = z * z$ eliminating the second, possible time-costly, evaluation. Disallowing side effects provides the referential transparency property, which makes it easier to verify, optimise, and parallelise programs. Although this is a very useful feature, most of the pure functional languages have been emphasised in academia rather than in commercial software development. Other features are recursion which is equivalent to iteration in imperative languages, dynamic typing and strict, non-strict and lazy evaluation which refer to how function arguments are processed when an expression is being evaluated. The first and still a very popular language in this area is the LISP programming language.

The LISP Language

The first functional-flavored language was LISP, developed by John McCarthy at MIT. Lisp was first implemented by Steve Russell on an IBM 704 computer. Russell had read McCarthy's paper [McC60], and realised that the *eval* function could be implemented as a Lisp interpreter which was a surprise for McCarthy [McC78]. Since that time many dialects have been derived from it most notably Scheme, Common LISP and Emacs LISP. The name LISP comes from "List Processing". LISP was originally created as an algebraic list processing language for artificial intelligence work based on Alonzo Church's lambda calculus. After its creation it became the favored programming language for artificial intelligence research. It is believed that LISP pioneered many ideas in computer science. Particular tree structures, garbage collection, automatic storage management, dynamic typing and the initial ideas of today's object oriented programming paradigm [JG93]. Lists are the main data structures of LISP. The LISP source code itself is made up of lists. Therefore LISP is called a "homoiconic" language where programs are represented as data structures. A program can manipulate source code as a data structure allowing programmers to extend LISP or even create their own languages. Languages with the feature of looking at their own source code are nowadays knows as reflexive languages. All program code in LISP is written as s-expression. A function call is a parenthesised list where the name of the function comes first followed by its arguments. The function application f with three arguments could be written like $(f\ x\ y\ z)$. Lisp utilises a dynamic type system which means the types are inferred and not explicitly written. This achieves run-time type flexibility and speed of program development by neglecting execution speed. To move towards the goal of achieving the safety and execution speed of traditional compiled languages, LISP encouraged the idea of type inference (e.g. [Bak90]). Type inferencing is the mechanical extraction of "type declarations" from a program. This information can be used by traditional compiled languages. An optimising compiler can then use this more precise type information to generate more efficient code. Although functional programming languages are not the most commonly used programming languages, there is no doubt that they founded many of the essential ideas of today's programming paradigms.

> *"Lisp has jokingly been called "the most intelligent way to misuse a computer". I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously*

impossible thoughts."

Edsger W. Dijkstra, The humble programmer, 1972

2.13 Concepts of Type Systems

This section discusses several concepts which have emerged in various programming languages and which are related to type systems:

Data Types The data types of type systems can be seen as a constraint of a variable which defines its domain. The domain is thereby the set of possible values which the variable is able to represent. It also defines how the data stored in a variable should be interpreted.

Interfaces of functions With a type system it is possible to define interfaces for functions. Such interfaces define precisely the parameter values and the return value of a functions. This significantly reduces the potential of passing wrong parameters to a function. Also the amount of necessary code for a program can be reduced as many error handling routines become obsolete with a type system.

Abstract types Abstract types are intended to protect internal program structures from unwanted external intervention. An abstract data type combines variables along with functions which operate on them. It is possible to allow only certain functions within the data type to be called by functions from the outside.

Polymorphism Polymorphism allows a function to handle data of different types with the same interface. Several types of polymorphism are distinguishable. They can be categorised as follows [Car91]:

- **Ad hoc polymorphism** allows a function to behave in different ways, depending on the type of a parameter.
- **Generic polymorphism** makes functions behave in a uniform way over all relevant types. Two subcategories can be identified:
  - **Parametric polymorphism** uses a type parameter or type variable instead of concrete variables with types. A generic written append function which is able to

connect all kinds of lists is an example of such polymorphism.

- **Subtype polymorphism** uses a subtype hierarchy and the fact that a subtype always embodies all features of its parent. A function written to handle a specific type can therefore handle also all its subtypes.

Subtyping Subtyping defines relationships between types which can be used to construct a hierarchy of types (see section 5.5 as an example). A subtype is usually more specialised than its parent, i.e. a subtype is a refinement of its parent. The ordering of types in this is very useful:

- Any subtype can be converted into its parent without losing any information.
- All operations which work on a specific type should also work with all its parents.

2.14 Characteristics of Type Systems for Programming Languages

Several properties can be used to characterise and evaluate a certain type system regardless whether it is from an imperative, functional or object oriented programming language.

2.14.1 Type Checking Strategies

There are principally two ways in which a program can be executed. Either the source code is compiled into machine language or the source code is directly executed via an interpreter. Both ways have advantages and disadvantages for the type checking. A statically typed language distinguishes between the compile-time and the run-time phases of program processing. All type checks are done during compile-time. This means that all types must be known when compiling the program. Furthermore, it is not possible to change the type of a variable during run-time. A programmer is enforced to think more about the types of his data structures. The result is clean source code. In production environments a statically typed language should be preferred [Pie02]. In a dynamically typed language the whole or parts of the type checking is done via runtime. These languages can be used for rapid prototyping to evaluate an idea or to quickly test a new approach - although also statically typed languages like Java have been used for this approach [Gos97]. Dynamic types languages appear more often in interpreted languages. They are

usually slower than compiled languages and should not be preferred for production environments. However, the possibility to influence and/or check the type of a variable during runtime offer many possibilities which are hard, if not impossible, to acquire in statically typed languages. Most interpreted languages have a type system but do not feature explicit type annotations. In these cases a type belongs to the value of the variable and not to the variable itself - types are in these cases sets of values [CW85].

2.14.2 Definition of Equality

One of the most important decisions when designing a type system is the definition of the equality. This property defines in which cases two objects are seen as equal. The equational theories vary widely from language to language. Most type systems can be categorised in one out of two cases being structural type systems and nominative (or by-name) type systems [Car04]. Many popular programming languages like C, C++ or Java have a nominative type system meaning that two types are equal if the declaration uses the same type name. In contrast some modern languages like Haskell or ML follow the structural type system approach whereas two types are equal if they have the same structure, meaning every feature within a type must have a corresponding or identical feature in the other type. A special form is the duck typing were the type is bound to the value of a variable and no longer to the variable itself. It can mainly be found in languages with dynamic typing and was mainly inspired by the Ruby language from Yukihiro Matsumoto [Mat01]. The languages emphasise the interfaces of an object rather than the specific types, well-designed code improves its flexibility by allowing polymorphic substitution. The standard example for duck typing in the Python language is the *file*-like classes. If a class implements some or all of the methods of the class *file*; it can be used everywhere where file would normally be used without implementing the interface of the *file* classes. Of course some objects have limitations and the duck typing cannot help where the use of an object does not make sense. The word duck typing comes from an aphorism: “If it waddles like a duck, and quacks like a duck, it’s a duck!”.

2.14.3 Safety of A Type System

Although types can also be used for performance tuning the safety aspect remains the main aim of a type system [Car04, ACPP91]. The design of the safety includes a trade off. The more safe a type system is, the less freedom remains for a programmer. In C for example the main philosophy was: "The programmer knows what he does" [KR88]. This implies that the programmer - or rather - his algorithm can do what he wants with the data in the memory. The Java language on the other hand is more restrictive and takes the aspect of encapsulation more seriously. The language is type safe and in most cases the exception handling system tells the programmer what has gone wrong [GM95]. The drawback is that this language cannot be used for every task (e.g. programming in the kernel space of an operating system like Linux or Windows). The advantage, however, is that no Java program is able to seriously harm its runtime environment due to ill-typed programming constructs. The main reason for C to be categorised as unsafe is the fact that it allows pointer arithmetic and thus give the programmer direct access to the memory [Moy92, Joy92]. In the white paper of Java the developers clearly underline the point by writing: "

> *The memory management model-no pointers or pointer arithmetic-eliminates entire classes of programming errors that bedevil C and C++ programmers.*"
>
> *J. Gosling and H. McGilton, The Java Language Environment[GM95], 1995*

2.14.4 Accuracy of A Type System

Another point which must be considered when defining a type system is its accuracy. A strong typed or memory-safe language is a language that does not allow undefined operations to occur. Consider the example:

```
x := 5;
y := "20";
print x + y;
```

Listing 2.1: Simple Code Example

Weakly typed languages would try to compute a result despite the different types of the operands for example 25 (Visual Basic), 520 (Java Script) or the result depends on the left-most operator

so $x+y=25$ while $y+x=520$ (Apple script). A strongly typed language on the other hand would simply reject those programs as ill-typed.

2.14.5 Scalability of Type Systems

Whether the type checking is still efficient when used in large scale software projects mainly depends on the type checking strategy and the defined complexity and flexibility of the type system itself. If a programming language uses only static typing, the size of a program does not matter at all because the type checking is performed before the program is compiled. After the static type checking is complete all type annotations are usually deleted which means that the type system is not used at all in the compiled executable. The drawback of course is that the abilities of statically checked type systems cannot be as complex and flexible as dynamically checked type systems (see section 2.14.1). Dynamically checked type systems on the other hand are very flexible and safe but can slow down the program significantly. Especially if the type system is very complex and many type checks and type conversions have to be done during runtime. Dynamic type systems can be very helpful in the first stage of a software project. However, when the system matures and comes into production environments a statically typed language should be preferred [Pie02].

2.15 Type Inference

The ability to simply infer the types of variables automatically makes many programming tasks easier. The programmer is left free to omit type annotations while maintaining type safety. This can be very helpful at the beginning of a project. In the prototype stage a developer just wants to quickly express his/her ideas and see if the project is at all feasible or not without worrying about accurate typing. For this “rapid prototyping” the type inference technique can be very useful. However, later when a project advances and grows to a complex software system it may be vital for it to allow only explicitly written typed annotations to obtain efficiency and more reliability [PS94]. Type inference is often very closely related to the type checking. The only difference is that the type checking algorithm tries to validate the declared types by means of the usage of the variables, while the type inference algorithm tries to infer the types from the usage of the variables. Or in other words instead of checking constraints the type inference generates and

records constraints for later consideration to infer the most general types for the variables [Pie02]. A common algorithm based on the typed lambda calculus for type inference is the Hindley-Milner or Damas-Milner algorithm. The origin of this algorithm dates back to 1958 to the type inference algorithm for the simply typed lambda calculus devised by Haskell B. Curry and Robert Feys. Roger Hindley extended their work in 1969 and proved that the algorithm will always infer the most general type. In 1978 Robin Milner independently provided an equivalent algorithm. In 1985 Luis Damas finally proved that Milner's algorithm was complete and extended it to support systems with polymorphic references. The complete algorithm can be found in [Pie02] (page 321 ff.). Another algorithm particularly designed for object oriented languages is presented in [PS91] and again published in the book [PS94]. This algorithm is interesting because it is simple, powerful and generally more related to the problems of programming languages in contrast to the approaches based on the lambda calculus. A common feature of all discussed approaches for type inference algorithms is that they operate in 2 steps:

1. Firstly they generate some constraints based on the untyped code they are analysing.
2. Secondly the algorithm tries to solve these constraints with the most general type possible.

2.16 Summary

This chapter reviewed and summarised related work for the FermaT transformation system and the presented approach of a Wide Spectrum Type System. It presented some critical aspects of current software engineering and explained why many of today's software products are unstable. The problems are not new and many reasons have been identified since the time of the software crisis. It is also known that no single solution in either technology or in management can by itself solve the problem. Huge software systems which are used in industry have usually a long development history. Most companies are afraid of changing their legacy systems over-night but keeping them too long is also dangerous as most of them run on old outdated hardware which is not produced anymore and are written in old programming languages which only a few programmers still know. The solution to this is software evolution which means to update or reengineer software systems continuously and adapt them to current business processes and technologies. However, any change in software system remains critical and needs approaches based on formal methods

to avoid the introduction of new errors or negative side-effects. A sophisticated solution for software reengineering and migration is the FermaT transformation theory. The approach is based on provably correct program transformations and has been researched since 1989. It has been implemented within the industrial-strength FermaT transformation system which uses its own specially designed intermediate programming language called Wide Spectrum Language. WSL is based on a small kernel language which itself is based on infinitary first order logic. Experience showed that a transformation theory without a strong formal foundation is prone to errors. WSL's denotational semantics define a program as a function from states (initial states) to sets of states (set of final states) which makes it possible to prove two programs as semantical equivalent even if their syntax is different. However, WSL and the migration process of FermaT have also several weaknesses due to the absence of a sophisticated type system. Type systems in programming languages have a long history which started with the Lambda Calculus. Today, they can be found in almost every current programming languages as they make a program more reliable by revealing inconsistencies in expression and procedure interfaces and, in case of explicit type systems, by forcing programmers to explicitly state their intention of usage for every variable. Over the recent years a huge variety of type systems with different type checking / inference strategies, definitions of equality, safety, accuracy and scalability have evolved. The presented approach uniquely combines many type system approaches into one coherent model to provide WSL with a suitable solution for its ability to represent programs on a wide spectrum of abstraction levels.

"Good judgement comes from experience, and experience comes from bad judgement."

Frederick Phillips Brooks, Jr.

Chapter 3

Preliminaries

Objectives

- Present how a programming language with a type system is constructed.
- Discuss and evaluate common approaches for the single construction steps.
- Describe the design decision which a designer of a programming language has to make.
- Introduce attribute grammars and their useful features for the type system.
- Illustrate the construction of a programming language with a practical example.

3.1 Introduction

This chapter gives an overview with references to later sections of the thesis about essential concepts which are needed to understand how the WSL programming language can be extended with the Wide Spectrum Type System. The chapter will elaborate how a programming language with a type system is constructed and present different approaches of how a type system can be used to identify malicious code. It will present and discuss ways of describing the semantics of a programming language. The chapter will also introduce concepts which can be used for type validation and the formal definition of the type system in general. Finally a small but complete example of a language definition will be presented to illustrate the presented concepts with a concrete example.

3.2 Composition of A Programming Language With A Type System

To fully understand what a type system can do and what the consequences are when types are introduced it is indispensable to review how a programming language is defined. In this context, it does not matter if the language is explicitly or implicitly typed. Normally a language is defined with 3 relations [Pie02]:

- The **syntactical** relation defines how the single atoms are named and how the terms can be constructed. This is commonly done via the Backus-Naur form (BNF).
- The **typing** relation constrains all the possible terms and sorts them into well-typed and ill-typed categories. Only the well-typed terms are considered as valid. This is commonly done via so called *inference rules* assigning types to terms (see section 3.8 for an example).
- The **semantic** relation at last defines how the terms may behave and how they should be executed. This can be done via operational, denotational or axiomatic approaches.

Coming from the simply typed lambda-calculus, two major ways of how a language definition can be organised have transpired in the past:

- The **Curry-style** first defines the terms of a language then the semantics are defined and finally a type system is defined that rejects terms whose behavior is unwanted.
- The **Church-style** on the other hand also first defines the syntactical relation but then it is extended already in the second step with a typing relation. The semantics are only defined for the remaining terms which have already been validated through the type system. Notable within this approach is that the question: "What is the behavior of an ill-typed term" will never arise.

Languages with an implicit type system are in most cases defined in the Curry-style while most explicit typed languages prefer the Church-style definition [Pie02].

3.3 Syntax of a Programming Language

The first step when defining a new language is to define the syntactical definition. This is done by defining grammatically how the single terms of a language can be constructed. This is similar to the grammar rules of a spoken language. Usually context-free grammars (CFG) are used to express the syntax for most of today's programming languages. A context-free grammar is a grammar which is defined through production rules of the form:

$$V \rightarrow w$$

V is a single non-terminal symbol, and w is a string of terminal and/or non-terminal symbols which can be empty. The term "context-free" refers to the fact that non-terminal symbols can be rewritten without regard to the context in which they occur. This allows every context-free language to be recognised by a non-deterministic push down automaton. A programming or formal language is context-free if they can be generated by a context-free grammar. The Backus Naur Form (BNF) was introduced by John Backus and Peter Naur to represent such grammars. In 1959 Backus presented a first approach of a description language which later evolved into BNF at the first World Computer Congress in Paris. The first version of BNF was created as part of creating the rules for Algol 60 [Bac60]. A BNF specification is a list of derivation rules, written as:

```
<symbol> ::= <expression with symbols>
```

Consider, for example, the number expression from Algol-60[1]:

```
<number>           ::= <unsigned number>
                       | + <unsigned number>
                       | - <unsigned number>
<unsigned number>  ::= <decimal number>
                       | <exponent part>
                       | <decimal number><exponent part>
<exponent part>    ::= #<integer>
<decimal number>   ::= <unsigned integer>
                       | <decimal fraction>
```

[1]This is taken from Appendix D of the CDC Algol-60 Version 5 Reference Manual © 1979 Control Data Corporation

```
                        | <unsigned integer><decimal fraction>
<unsigned integer> ::= <digit>
                        | <unsigned integer><digit>
<decimal fraction> ::= .<unsigned integer>
<digit>            ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A serious drawback was that the BNF was not able to express options and repetitions could not be directly expressed. This causes many definitions to be unnecessary long and complicated to understand. This motivates the definition of the EBNF originally developed by Niklaus Wirth [Wir77]. The extension included option `[]`, repetition (possibly empty) `{}` and group `()` while the brackets `<>` could be omitted. Terminals in EBNF are strictly enclosed in quotation marks. The example above would look like this when written in EBNF:

```
number           ::= [ "+" | "-" ] unsigned number
unsigned_number  ::= decimal_number [ exponent_part ] | exponent_part
exponent_part    ::= "#" integer
decimal_number   ::= unsigned_integer [ decimal_fraction ] | decimal_fraction
unsigned_integer ::= digit | unsigned_integer digit
decimal_fraction ::= "." unsigned_integer
digit            ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Nowadays many more extensions of the BNF are available to express nearly everything. One common feature of many variants is the use of regexp wild-cards such as * and +. The example above expressed in an EBNF including regexp wild-cards would look like this:

```
number           ::= [ "+" | "-" ] unsigned number
unsigned_number  ::= decimal_number [ exponent_part ] | exponent_part
exponent_part    ::= "#" integer
decimal_number   ::= unsigned_integer [ decimal_fraction ] | decimal_fraction
unsigned_integer ::= digit+
decimal_fraction ::= "." unsigned_integer
digit            ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Nowadays the Backus Naur Form can be seen as the de-facto standard for specifying the syntactical relation of programming languages. It led also to powerful software tools know as Compiler-Compiler. These tools are able to “compile” a language parser out of a BNF-like definition. The

EBNF is currently defined by ISO standard ISO/IEC 14977:1996(E) [Int96].

3.4 Abstract Syntax Tree

As the name suggests the abstract syntax tree abstracts from the source program the syntactic structure, without the spelling of particular identifiers, keywords, comments, spaces and line breaks. The Abstract Syntax Tree is generated from the Parse Tree by omitting nodes which have no semantic meaning [PP92]. Consider, for example, the program:

```
IF x = 0
  THEN y := 1
  ELSE PRINT("Goodby cruel world") FI
```

Listing 3.1: Example Code

The first step is to split up the whole program into distinct tokens called Lexer Tokens: According

Program	Token	Token Value
IF	S_IF	
x	S_IDENTIFIER	x
=	S_EQUAL	
0	S_NUMBER	0
THEN	S_THEN	
y	S_IDENTIFIER	y
:=	S_BECOMES	
1	S_NUMBER	1
ELSE	S_ELSE	
PRINT	S_PRINT	
(	S_LPAREN	
"Goodby cruel world"	S_STRING	Goodby cruel world
)	S_RPAREN	
FI	S_FI	

Figure 3.1: Example Lexer Tokens

to the syntactical definition of a language, a parser can now generate the Parse Tree (see figure 3.2). This tree is rather huge and difficult to process. The actual Abstract Syntax Tree (AST) therefore omits all nodes which are semantically not relevant (see figure 3.3). This makes the processing of the tree a lot easier and faster. The “if” represents a conditional statement “**Cond**” which has two guards. The code behind a guard node is only evaluated under a certain condition. The code after the first guard requires that x is equal to 0. If this is not the case the second guard will be evaluated. The code behind the second guard will always be executed when the guard is evaluated because the guard condition is always true. At a first glance the tree might

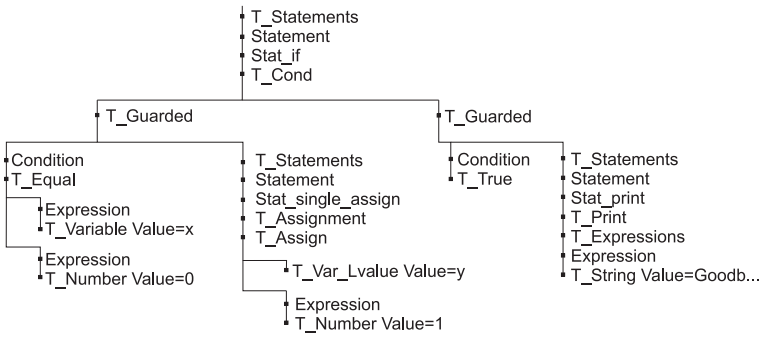

Figure 3.2: Example Parse Tree

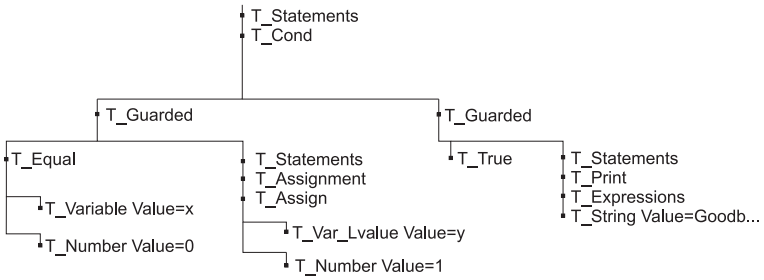

Figure 3.3: Example Abstract Syntax Tree

look more complex and indeed it is definitively harder to understand for the human. But looking closer from the perspective of a machine, it is much easier to traversal a tree structure than a sequence of characters. Furthermore, the tree represents a data structure which can be used to store additional information (e.g. constrains for type checking). In past decades the theory for language parsing and compiler design has been the focus of many research projects. No other area in computer science has been so extensively discussed than programming language and compiler design [PP92].

3.5 Typing of a Programming Language

The main purpose of type systems for programming languages is to prevent the occurrence of execution errors during the running of a program [Car04]. While types are usually not very helpful in hand proofs, they do help with mechanised proofs of correctness [LP99]. Many ways have been developed to introduce type systems in various formal systems. However, many compilers of programming languages tend to just hard code the type checking, omitting a formal definition. Mostly functional and some object oriented programming languages tend to have a sound formal

foundation and thus also a sound formal definition of a type system. A type system consists of defined data types and typing rules that are used to assess the type correctness. If all expressions of a program obeys these rules then the program can be considered as type safe. Each typing rule has the following form (see section 4.4 for the formal definition):

$$\frac{\Gamma_1 \vdash \Im : T_1 \ldots \Gamma_n \vdash \Im : T_n}{\Gamma \vdash \Im : T} \qquad \begin{array}{l} \textit{Premise Judgment} \\ \textit{Conclusion Judgment} \end{array}$$

Each typing rule has one or more premise judgments and one conclusion statement. The conclusion statement must match an evaluated expression before the typing rule is applicable. With every applicable typing rule the conclusion judgments must hold if all premise judgments are true [Car04]. In 1968 Donald E. Knuth developed attribute grammars which can be elegantly used for expressing type checking rules (see section 4.4).

3.6 Semantics of a Programming Language

To be able to use a programming language it is necessary to define precisely how a particular term should be evaluated. Pierce et al. [Pie02] identified three major approaches to formalise this so called semantic relation of a language.

1. Operational semantics
2. Denotational semantics
3. Axiomatic semantics

A clean distinction between the different approaches is not always possible, but all known approaches to formal semantics of programming languages use the above techniques, or some combination thereof.

3.6.1 Operational Semantics

The operational semantics approach is maybe the most simple but also the most flexible approach of the three. The semantics are defined by an abstract machine which uses the terms of the

language as input. The behavior is defined by a transition function for each of the states which either gives the next state or halts the machine (stuck state).

3.6.2 Denotational Semantics

Denotational semantics capture the behavior of terms in a more abstract view then the operational semantics. At first the approach defines a number of semantic domains. The second step is to define interpretation functions which map terms into elements of these domains. The major advantage of this approach is that it does not get lost in the details of evaluation. It merely abstracts the essential concepts of the language. The properties of the chosen semantic domains can be used to derive powerful rules for reasoning about the behavior. These rules can prove that two programs have the same behavior which is called "semantical equivalent" or that a given program satisfies some specification. Section 2.7.3 describes the definition of WSL which is based on denotational semantics.

3.6.3 Axiomatic Semantics

This form of semantic definition is the most mathematical approach. Instead of first defining the behavior and then afterwards deriving rules from it this approach tries directly to formalise the semantics by rules; the so called axioms. The advantage of the approach is that the focus is from the beginning on the process of reasoning about programs. The disadvantage is of course the time consuming definition and the complicated and complex form that is not as easy to understand as the other approaches.

3.6.4 Other Variants of Formal Semantics

Many specialised approaches can be found to formalise the semantic meaning of a programming language. Some very common ones are:

1. Algebraic Semantics

   Algebraic semantics of a programming language is a form of axiomatic semantics based on

algebraic laws. The basic idea of the algebraic approach to semantics is to identify and name different sorts of objects and to associate operations with these sets of objects. Algebraic axioms are then used to describe the properties.

2. Action semantics

   This approach tries to split the formalisation process of denotational semantics into two layers (the macro- and micro semantics). To simplify the specification it predefines three semantic entities (actions, data and Yielders). Action semantics are a mixture of denotational, operational and algebraic semantics.

3. Attribute grammars

   Besides the use of attribute grammars described in section 3.7, they can also be used to formulate semantic meaning. Attribute grammars can be understood as denotational semantics where the target language is the original language extended with attribute annotations.

Some other approaches which will not be described further are: Categorical (or "functorial") Semantics, Concurrency Semantics and Game semantics.

3.7 Attribute Grammars

A fundamental step in computer science was the invention of attribute grammars by Donald E. Knuth in 1968 [Knu68] [Knu90]. An attribute grammar extends the productions of a formal grammar with attributes. With these attributes it is possible to restrict the set of syntactical correct strings of a particular language to those that meet certain semantic constrains which is ideal for a type system. For example: The declared or inferred type of any variable or sub-expression must be consistent with its use. Formally it can be seen as a triple:

$$G_A = (G, A, F)$$

Consisting of a context-free grammar G, a finite set of attributes A and a finite set of attribute assertions, or predicates F about the attributes. The attributes can be categorised into two types. The *synthesised attributes* are computed according to some evaluation rules while the *inherited attributes* are computed from parent nodes. These types of attributes can be used to model a data

flow of attribute information which can be very useful when the assertion of a tree node needs an attribute value from a sibling node. Synthesised attributes must be processed first as inherited

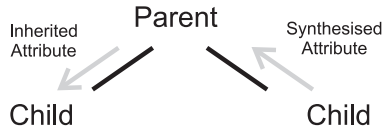

Figure 3.4: Data Flow Between Inherited and Synthesised Attributes

attributes depend on them. Each attribute and assertion is associated with a single non-terminal or terminal of the grammar. Each assertion refers only to those attributes associated with its grammar production - although it is possible to define some functions for the assertions which can be used to access global data structures (e.g. a symbol table for variables to check if a variable is declared or to get its declared type). A string in the language G is also in the language G_A iff all applicable assertions f (e.g. typing rules) hold true for all applicable nodes.

$$G \Rightarrow G_A \Longleftrightarrow \forall f \in F.f = true$$

Consider the simple expression language:

```
condition           ::= expression condition_operator expression
expression          ::= value operator value
operator            ::= boolean_op | integer_op
boolean_op          ::= "OR" | "AND"
integer_op          ::= "+" | "-"
condition_op        ::= "="
number_condition_op ::= "<" | ">"
value               ::= boolean | number
boolean             ::= true | false
number              ::= digit+
digit    ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

This language can be extended with the attribute “type”. The prefix “^” defines a synthesised attribute while “!” defines an inherited attribute. Because the evaluation will only be performed on semantically relevant nodes, only nodes which are used in the Abstract Syntax Tree have assertions. The different names for the attributes inside a production are only introduced to distinguish each of them for the assignment. In the Abstract Syntax Tree every tree node will have only one attribute:

```
condition(^t1)   ::= expression(^t2) condition_operator(!t3) expression(^t4)
   [t2 = t4 & t2 = t3 & t1 = t2 & t3={boolean}]

expression(^t1)  ::= value(^t2) operator(!t3) value(^t4)
   [t2 = t4 & t2 = t3 & t1 = t2]

operator(!t)            ::= boolean_op(!t) | integer_op(!t)

boolean_op(!t)          ::= "OR" | "AND"          [t = {boolean}]
integer_op(!t)          ::= "+" | "-"             [t = {integer}]
condition_op(!t)        ::= "="                   [t IN {boolean,integer}]
number_condition_op(!t) ::= "<" | ">"             [t = {integer}]
value(t)                ::= boolean(t)
                            | number(t)
boolean(^t)             ::= true | false          [t = {boolean}]
number(^t)              ::= digit+                [t = {integer}]
digit    ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

In the *expression* production the synthesised value of attribute *type2* is passed down via the inherited attribute *type3* to the production *operator* which passes the value further down to *boolean_operator* or *integer_operator* which than checks if the operands have the correct type. Therefore the tree has to be evaluated bottom-up and each node having nodes with inherited attributes as children must be evaluated top-down a second time. Consider for example the small program:

```
5 + 3 = 4 + 4
```

The corresponding Abstract Syntax Tree would look as in figure 3.5 (to trace the data flow the "type" attributes are numbered according to their usage and the flow direction is marked). Besides all their advantages, attribute grammars naturally also have some drawbacks. The handling of non-local information is not trivial and depends on the definition of special functions for the assertions as well as a proper attribute evaluation strategy (e.g. left-right evaluation) in the tree. Another difficulty with very complex attribute grammars is to avoid a possible circular evaluation. Section 4.4.5 shows an example of how a type checker can use the previously mentioned attribute grammars to efficiently check a given program against typing rules.

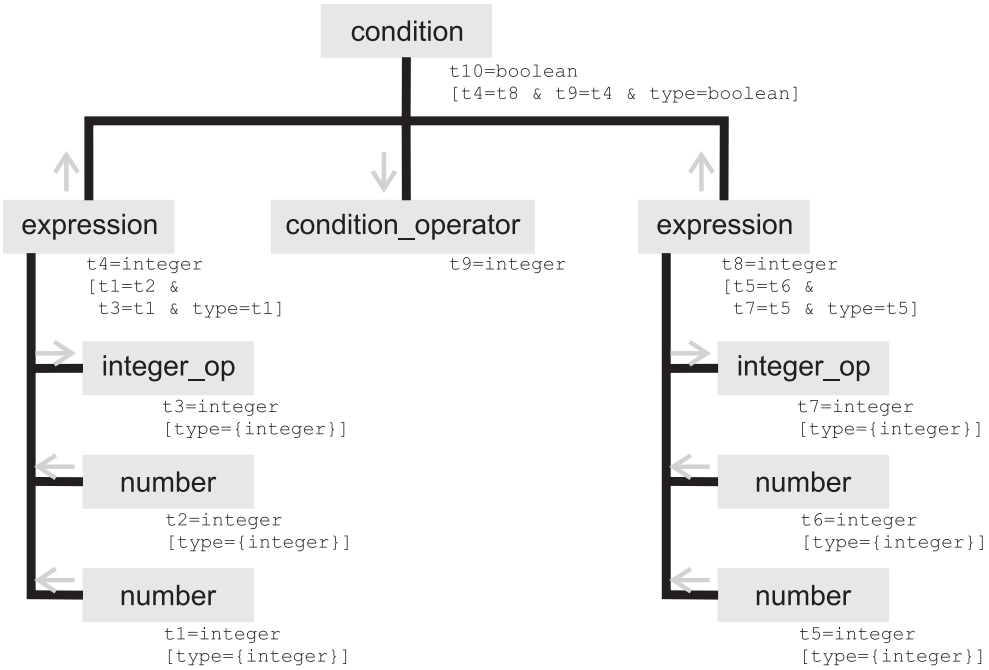

Figure 3.5: Abstract Syntax Tree With Attributes

3.8 Example of a Language Definition

To show the possibilities of typed languages this section introduces a brief example of a language definition containing only boolean and number expressions. Four syntactically correct programs are presented of which two would end up in a stuck state, i.e. run-time error. Notably, these programs can be rejected before execution by a type system as ill-typed.

3.8.1 Syntactical Definition:

```
term    ::=   boolean
            | "if" term "then" term "else" term
            | number
            | "iszero" "(" term ")"

zero    ::= "0"
number  ::= ("1" | "2" | "3" | "4" | "5"
             | "6" | "7" | "8" | "9") (number | zero)*
boolean ::= true | false
```

It is already possible to write programs. But to be able to evaluate these programs the semantic relation for the language is required.

3.8.2 Semantic Definition (Operational Semantics):

$$if\ true\ then\ t_2\ else\ t_3\ \rightarrow\ t_2$$

$$if\ \ false\ then\ t_2\ else\ t_3\ \rightarrow\ t_3$$

$$iszero(zero) \rightarrow\ true$$

$$iszero(number) \rightarrow\ \ false$$

$$\frac{t_1\ \rightarrow\ t_1'}{if\ t_1\ then\ t_2\ else\ t_3\ \rightarrow\ if\ t_1'\ then\ t_2\ else\ t_3}$$

$$\frac{t_1\ \rightarrow\ t_1'}{iszero(t_1) \rightarrow\ iszero(t_1')}$$

With this definition it is now possible to write a compiler which can translate programs into machine code. Consider, for example, the following programs:

```
(1) if iszero(0) then true else false

(2) if false then 0 else 1

(3) if 1 then true else false

(4) if true then iszero(true) else false
```

Listing 3.2: Example Programs

The first program would evaluate to true in the following derivation tree:

$$\frac{}{iszero(0)\ \rightarrow\ true}$$

$$\frac{}{if\ iszero(0)\ then\ true\ else\ \ false\ \rightarrow\ if\ true\ then\ true\ else\ \ false}$$

$$\frac{}{if\ iszero(0)\ then\ true\ else\ \ false\ \rightarrow\ true}$$

The second program would evaluate 1 and the third and fourth, although syntactically correct,

would end up in a stuck state as their evaluation is not defined by the semantics. The programs 3 and 4 are therefore meaningless or erroneous programs. The disadvantage of the current language definition is that the errors are only detected when the programs are evaluated. In large scale programs with many branches it is very difficult and in many cases impossible to test the evaluation of all possible branches. Fortunately, type systems can be used to reject many meaningless programs without the need of evaluation.

3.8.3 Type System Definition:

As the language contains only boolean and number expressions it is sufficient to only define two types:

```
T ::=   bool | int
```

The type system itself is defined by the following typing rules:

$$\frac{}{true, false : bool}$$

$$\frac{}{0,1,2,3,4,5,6,7,8,9 : int}$$

$$\frac{t_1 : bool \quad t_2 : T \quad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3\ : bool}$$

$$\frac{t_1 : int}{iszero(t1)\ : bool}$$

The primitive values are either boolean or int. The first expression in an “if” statement must be of type boolean while its branches can have any type. An “if” statement in this case returns the value and type of the expression which is evaluated. The “iszero” has an argument of type integer and the statement returns always a value of type boolean. To check and verify the correctness of typing a “type derivation” tree is constructed which consists of concrete instances of typing rules [Pie02]. A program is considered as type safe if such a tree can be constructed for every one of

$$\frac{\dfrac{\texttt{0}\textbf{: int}}{\texttt{iszero(0)}\textbf{: bool}} \qquad \dfrac{}{\texttt{true}\textbf{: bool}} \qquad \dfrac{}{\texttt{false}\textbf{: bool}}}{\texttt{if iszero(0) then true else false}\ \textbf{:bool}}$$

Figure 3.6: Example Type Derivation Tree

its expressions. Figure 3.6 shows the "type derivation" tree of program 1. With the aid of typing rules, it is possible to reject the third and fourth program before evaluation as ill-typed because the "type derivation" cannot be constructed, i.e. a sequence of inference rules which defines the type cannot be found. If the type system is not too complex and all variable types are known a good type checker can fairly quickly distinguish whether a term is ill typed or not.

3.9 Summary

This chapter gave an overview about essential concepts for the construction of programming languages such as the definition of the syntax via BNF, the construction of a type system with the definition of data types and typing rules as well as the definition of semantics via operational, denotational or axiomatical approaches. Furthermore, it introduced attribute grammars which can add attributes and attribute assertions to an already defined context-free grammar. Attribute grammars will help in the next chapter to formally defines the type system around WSL and to validate the type correctness of a given program. The chapter finalised with an example construction of a simple programming language by defining its syntactical, semantic and typing relation.

"*Computer Science is no more about computers than astronomy is about telescopes.*"

Edsger Wybe Dijkstra

Chapter 4

Wide Spectrum Type System

Objectives

- Define the key features of the Wide Spectrum Type System.
- Introduce the type system to WSL formally by using attribute grammars.
- Describe the process of type checking and type inference.

4.1 Introduction

This chapter gives an initial introduction into the Wide Spectrum Type System. It will at first discuss its main features which are the layered approach, the emphasis on explicit typing, the awareness of variable precision and storage size and scalability. The chapter will then formally introduce the type system into the WSL programming language by using attribute grammars. It will conclude by explaining how the type checking and type inference algorithms work.

4.2 Approach of the Wide Spectrum Type System

The object oriented programming paradigm has demonstrated that the introduction of defined interfaces using data types between distinct software components is a major step towards main-

taining software quality [PS94]. Data types should define explicitly the possible values and possible operations of a variable. They should prevent a program from performing illegal operations which may cause the program to abort unexpectedly. In an explicitly typed programming language the declaration of a variable can be seen as an invariant assertion. A type checker can check the usage of a variable throughout the program against this invariant assertion. Type checking can be seen as the mechanisation of a mathematical proof. A successful type check, especially in strongly-typed languages is like a certificate for the program verifying that it is free of certain types of errors. Almost all of today's commercially relevant programming languages use data types to reduce the amount of introduced errors during the development process. Although data types are used for clarification they do not have a clear and generally standardised definition themselves. However, two fundamental properties can be identified when looking at the majority of commercially relevant static typed programming languages [Pie02]:

- Weakly / Strongly typed: A strongly typed language does not allow an operation to succeed on arguments which have the wrong type while weakly typed languages provide the programmer with much more freedom to use any variable as desired directly through the use of pointers or by type cast.
- Safe / Unsafe typed: A language is type-safe if it does not allow operations or conversions which might cause an abnormal termination of the program. Unsafe typed languages on the other hand rely more on the programmer to realise a reliable and sophisticated implementation giving him the freedom to program in unusual ways, for example, to enhance performance.

Almost every programming language has its own understanding of data types and in many cases certain features of the type system are decided by the concrete compiler when the executable is generated (e.g. by obeying certain command line options). This means that in some cases the concrete definition of a data type depends not only on the chosen programming language but also on the used compiler. Nevertheless, this diversity of type systems can be very helpful for developers. For every single software project a developer can choose an appropriate programming language which provides as much security as required whilst preserving as much flexibility as possible. Unfortunately, the advantage of a high diversity of type systems becomes a problem

when an important software program or, even more seriously, its programming language becomes older and eventually a legacy. The reengineering or migration of these systems is very hard because of the implicit implications and assertions which come with data types. As discussed in section 2.10 also the FermaT migration process has several flaws in the absence of a sophisticated type system. To tackle these flaws the Wide Spectrum Type System approach was developed to take the diversity of type systems found in current legacy applications as well as potential migration target languages into account. The approach puts emphasis on commercially relevant programming languages:

- like FORTRAN 77, COBOL and C which were utilised for the majority of legacy systems as potential source languages.
- like C, C++, FORTRAN 95, Java and C# which are currently the most reasonable migration targets as potential target languages.

The approach is designed to have the following key features:

- Layered Approach
- Emphasising explicit typing
- Awareness of Varying Precision and Storage Size of Variables
- Scalability

4.2.1 Layered Approach

Almost all programming languages are designed for a special purpose. Therefore their possible programming constructs and data types differ significantly. Some languages allow unsafe constructs like pointers (e.g. C) to give as much possible freedom to the programmer and to allow very hardware related programming [KR88] while the fundamental philosophy of other languages prohibit particular those unsafe constructs in order to guarantee a high-level of safety (e.g. Java [GM95]). Despite the philosophy also the range of possible data types differs a lot from language to language. Because of this diverseness of approaches and philosophies an intermediate language

which features only a single type system approach is always insufficient. The solution presented in this thesis therefore proposes a layered type system. Figure 4.1 shows the layered approach in more detail. Each layer represents a separate type system with a distinct set of data types and a certain level of security i.e. level of strictness.

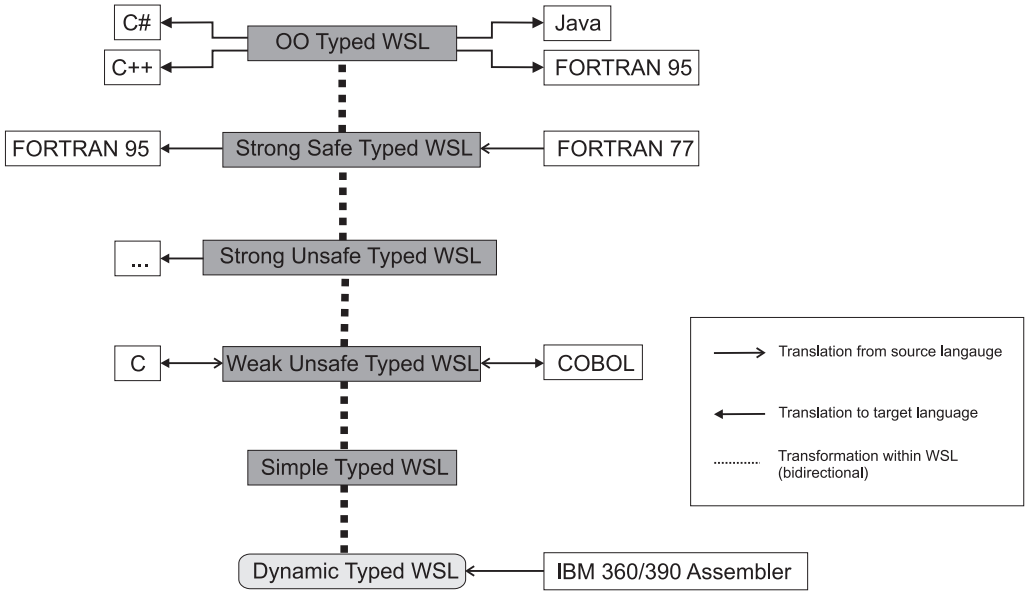

Figure 4.1: Layered Wide-Spectrum Type System

- **Dynamic Typed WSL** represents the current untyped WSL.
- **Simple Typed WSL** represents the current WSL with explicit typing. This layer is mainly used if an untyped program should be typed through type inference.
- **Weak Unsafe Typed WSL** is used to represent weakly typed languages like C which feature pointers and allow any form of type casting.
- **Strong Unsafe Typed WSL** focuses on languages which have a stronger typing. Although unsafe constructs like pointers are still allowed, a type cast can only perform "upcasting" operations which guarantees that no information gets lost. Any other alteration of typing has to be done through special conversion functions.
- **Strong Safe Typed WSL** is like the previous layer except for the disallowance of direct memory access through pointers.

- **Object Oriented Typed WSL** represents an WSL with Object Oriented typing. A procedural program on this layer can be converted step-by-step into an Object Oriented program.

Most common procedural programming languages can be translated to and from WSL using one of these layers. Although the Wide Spectrum Type System introduces an object oriented layer, it is far less expressive and flexible as other state-of-the-art object oriented languages such as Java, for example. Instead, it offers a carefully defined mixture of procedural and object oriented facilities by avoiding to introduce new semantic definitions. The intention of this layer is to ease the migration from procedural into object oriented code by preserving the procedural structures and wrapping the object definition around them[1]. All other layers, however, are as expressive as their intended source / target languages. Special Type System Transformations can be used to raise or lower the level of strictness (see section 6.5). If a program uses constructs which are illegal on the destination layer, the maintainer can either apply some of FermaTs semantic preserving code transformations or carefully alter the source code directly. Unfortunately, the usage of pointers still creates many difficult problems which may hinder a successful transition among the safely typed layers.

4.2.2 Emphasising Explicit Typing

A critical issue in any migration process is the implicit conversion of data types. If a type needs to be converted during an arithmetic expression it is common practice to convert the type of the variable implicitly. In fact this makes the code shorter and normally easier to read but it has great potential to corrupt the processed data. Even if the migration process is successful and the translation is accurate, these conversions may cause errors as the reengineered system might have to operate on different input data. An infamous example of such an error is the flight of the first Ariane 5 space rocket which was launched on June 4, 1996. The Ariane 5 software reused the specifications from the Ariane 4, despite the fact that the Ariane 5 flight path was significantly different and beyond the range for which the reused code had been designed. Especially, the much greater acceleration of the Ariane 5's caused the primary and back-up guidance computers to crash after the start. This caused a misguidance of the launcher's nozzles by false data, critically misleading the missile. This led to a cascade of problems, culminating in the self-destruction of the

[1] Section 5.11 and 5.3.14explains how common object oriented features can be represented in the Object Oriented Typed layer without introducing new semantics to WSL.

entire rocket 37 seconds after launch. Luckily, there was no human loss but with costs of around 0.36 Billion USD it is one of the most expensive computer bugs in history. The initial cause for this disaster was the unsafe design involving a data conversion between 64-bit floating point to 16-bit integer. Although the run-time environment would have been able to cope with such errors, it was decided to turn these features off due to efficiency considerations [Eur96, Lan97]. To help identify any potentially dangerous constructs, the type checker of WSL enforces the explicit expression of every type cast within a program. It, furthermore, disallows any type casts which involve a loss of precision in its strongly typed layers.

4.2.3 Awareness of Varying Precision and Storage Size of Variables

Most programming languages leave out the fact that a machine will always be finite while mathematics are generally not. The data type `int` for example will always be just an approximation of the mathematically set of integers Z whereby the accuracy of this approximation highly depends on the amount of allocated memory. A 4 byte long signed integer for example would be able to represent any value between -2.147.483.648 to 2.147.483.647 while all attempts to express a number outside this range will result in an overflow. Another example is the data type `float`. A float should approximate the algebraic set of real numbers R which is not only infinite but uncountable infinite. Unlike integers, which are either precise or, in case of an overflow, totally wrong, a floating point numbers is within its limits not precise due to rounding errors. Consider the irrational number PI expressed in 4 byte, 8 byte and 16 byte precision:

PI	**Precision**
3.14159265358979323846	16 byte
3.14159265358979310000	8 byte
3.14159274101257320000	4 byte

While the first digits of the number are always correct, the mantissa differs significantly towards the end. Unfortunately, it is unpredictable where exactly the inaccuracy begins as it is highly dependent on the represented value. This means that a check for the correctness of a result is almost impossible and the preserving of data precision is one of the major challenges during the migration process. If a legacy system is largely scaled and the target platform defines the lengths

for several data types differently, the trace of resulting errors can be like looking for a needle in a haystack. The problem gets even more serious if a programming language does not have a standardised length for data types. In C, for example, the length of a data type is decided by the concrete compiler which builds the binary executable code. An example of an extreme difficult code migration is to port a large scale C program from a Cray SV1 to an IBM machine with AIX as operating system. Although the process does not involve any code translation, the data types "short", "int" and "float" would lose at least 4 bytes of precision in any case. Furthermore, it would lose by default at least 4 bytes for the data types "long" and "long double" unless the "xlc" compiler switches "-q64" and "-qlongdouble" are used [Int05]. Another difficulty arises when migrating between platforms which have different pointer sizes to address their memory. This is the case when migrating from old 16 bit CPUs to standard 32 bit CPUs or from any 32 bit assembler or C to AS400 Cobol which uses 128 bit pointers [Int94]. Examples like these show that a clear and precise definition of data types is an essential precondition for any successful evolution of software. Because of WSL's aim to operate as an intermediate language its Wide Spectrum Type System puts a strong emphasis on a clear and unambiguous storage definition by including the length of a data type in its definition. For example:

- A list of 5 elements starting with an index of 1 is declared as `LIST[1:5]`.
- A list with a dynamic range can be declared as `LIST[1:*]` or `LIST[0:*]`.
- A variable with a known storage size is declared with its type, followed by a star, followed by its storage size e.g. a 4 byte integer is declared as `INTEGER*4`.
- A variable whose storage size is not known can be denoted as `<TYPE>*0` e.g. inferred types or variables with variable storage size like STRING in Java. Only in this case the platform and language specific default storage size of its data type is used.

4.2.4 Scalability

Another important requirement for any practical approach is its scalability. The scalability within this approach means mainly the performance of the type system verification process and the type transformations. The runtime of a processed legacy system, however, is never affected as the Wide

Spectrum Type System uses only static typing which means that all type information of a program is deleted when it is translated into machine language for execution. The verification process and the type transformations on all type system layers should remain efficient and yield results within a reasonable time when applied on large scale systems. The following design decisions were made to achieve this:

- The definition of the types is simple and flexible as more complicated definitions usually cause a large overhead during verification and tend to be more error prone.
- Type checks are localised and involve as few elements as possible. Therefore, only expressions are checked and not whole statements.
- The type checker stops immediately after a typing error is encountered. This reduces the check-modify-recheck cycle and avoids the identification of follow-up errors which might be caused by a previous error.
- The machine readable typing rules are organised in hash tables. This technique assures that only relevant typing rules are used when checking expressions of a program.

Of course the phrase *reasonable time* is very abstract and ambiguous. Therefore the type checking and type inference algorithms for this approach were implemented and tested (discussed later in section 7.4). A performance tests which succeeded in *reasonable time* was the type inferencing and type checking of a large WSL module consisting of around 9000 lines of code with about 180 variables. A full type check took 1 minute while the type inferencing algorithm needed only 40 seconds. The reason for this is that a type check runs through the code for each variable separately while the type inference algorithm loops only until no new types can be inferred. The implemented type check is linear to the code size and number of variables while the time for type inference depends also on the code itself i.e. used constants and operators in formulas as well as dependencies between variables.

4.3 Introduction of the Wide Spectrum Type System to WSL

The extension of a theoretical model, in this case the language definition of WSL, has to be done very carefully so as to avoid accidentally introducing ambiguities or inconsistencies. In terms of the Wide Spectrum type system the most important question is: What is the actual effect of the Wide Spectrum Type System on WSL and the current set of code transformations? Obviously the introduction of a type system to WSL would be very difficult if it changed some of its semantics. In some circumstances this could even mean that many of FermaT's transformations would have to be proven again as the current proofs for the correctness of these transformations are based on the untyped version of WSL [War04]. To avoid such steps the typing information is exclusively taken from the legacy system itself or, if the legacy system does not feature a type system, the type inference algorithm is used to capture implicit typing and make it explicit. Inferred types, in this sense, can be seen as assertions which only reveal what is already implicitly stated inside a program. An inference error is concluded if a variable is not used consistently throughout the code. Furthermore, the suggested integration into the transformation process places the introduction of the type system into WSL after the transformations have been applied (see section 7.4.5). In fact, the Wide Spectrum Type System is only to be used to validate the typing of a transformed legacy system against the typing rules of the targeted programming language.

4.4 Mathematical Foundation

The Wide Spectrum Language (WSL) is enhanced with a Wide Spectrum Type System by extending the normal grammar of WSL to an attribute grammar. Attribute grammars were firstly mentioned by Donald E. Knuth in 1968 [Knu68] while the proposed approach was derived from [PP92]. An Attribute grammar extends the productions of a formal grammar with attributes. With these attributes it is possible to restrict the set of syntactical correct strings of a particular language to those that meet certain *semantic constraints*. In a type system such a constraint could be: The declared or inferred type of any variable or sub-expression must be consistent with its use. The advantage of this approach is that the *semantic constraints* i.e. typing rules can be defined, *outside* the current semantic definition of WSL, for each layer of the layered type system

separately. This means that the behavior of the type checker can be easily adjusted by changing the operational set of typing rules. Another advantage is that the algorithms for type checking and type inference are very similar and can operate on the same set of constraints.

4.4.1 Attribute Grammar for WSL

The attribute grammar of WSL is a triple: $A = (G,V,F)$ consisting of the context-free grammar G of untyped WSL with added productions for typing[2], a finite set of distinct attributes V which in this case always represent types, and a finite set of assertions F about the attributes known as typing rules. Each attribute is associated with a single non-terminal or terminal of the grammar and each typing rule is associated with a single production. A string which is valid in the language of G is also valid in the language of A *iff* for every attached attribute of every terminal- and non-terminal node of the abstract syntax tree at least one assertion holds true.

4.4.2 Typing Judgments

The assertions F are described by type judgments. A typical judgment has the form:

$$\Gamma \vdash \Im : T$$

Here Γ is a *static type environment* and $\Im$ is an assertion. It is defined that Γ *entails* $\Im$. The type environment is an ordered list of distinct variables and their types. The empty environment is denoted $\oslash$. All free variables of $\Im$ must be declared in Γ. Within the Wide Spectrum Type System a judgment will assign the type T to the terminals or non-terminals of their assigned production.

4.4.3 Typing Rules

Typing rules assert the validity of certain judgments on the basis of other judgments that are already known to be valid. The judgment whose validity should be asserted is known as conclusion

[2]This is a necessary syntactical extension to enable the user to write data types when declaring variables or type casts when converting types.

judgment. Some production might have more than one judgment associated. Therefore, each judgment has one or more premise judgments which must be true before the validity of the conclusion judgment is checked. If the premise judgment is empty the conclusion is true for all applicable expressions. The optional premise judgments together with the conclusion judgment form a typing rule:

$$\frac{\Gamma \vdash \Im : T_1 \ldots \Im : T_n}{\Gamma \vdash \Im : T} \qquad \begin{array}{l} \textit{Premise Judgment} \\ \textit{Conclusion Judgment} \end{array}$$

4.4.4 Correctness of Algorithms

Given the set of functions of all type transformations and object identification algorithms Ω, the set of possible WSL programs Σ, the set of applicable typing rules Δ and the type check function C.

$$\begin{array}{rl} let & T \in \Omega \textit{ and } P \in \Sigma \textit{ and } R \in \Delta \\ forall & R, T, P : T(P) \approx P \\ iff & C(P) \wedge C(T(P)) \wedge R \end{array}$$

The correctness and accuracy of the type checking and type inferencing depends mostly on the typing rules which are the axioms of the formal type system. Each layer of the Wide Spectrum Type Systems has over 100 single typing rules which need to be correct. So far, their correctness was verified by applying them on various case studies (see chapter 8) and code examples including large quantities of FermaTs own WSL source code.

4.4.5 Example

The following example will demonstrate the possibilities of attribute grammars in combination with a type system with a simple language consisting of only two operations and 4 values. During the example the grammar of the language will be enhanced with attributes to perform type checking on two programs. Let the given context-free grammar G be:

$$T \rightarrow T + T \mid T \textit{ OR } T \mid \text{"0"} \mid \text{"1"} \mid \textit{"TRUE"} \mid \textit{"FALSE"}$$

According to this definition the following programs would be valid:

1) $1 + 0 + 1$

2) $1 + 0 + \mathit{FALSE\ OR\ TRUE}$

Now, to enhance the grammar of the language, an attribute "type" which can be either INTEGER or BOOLEAN is defined for each terminal and non-terminal statement of the grammar. These attributes form a finite set of distinct attributes V and will enhance any derived Abstract Syntax Trees with extra information. To restrict the set of syntactical correct strings (programs) of the simple language a set of assertions (typing rules) F is defined:

$$\frac{\Gamma \vdash A{:}\mathit{BOOLEAN}\ \ \Gamma \vdash B{:}\mathit{BOOLEAN}}{\Gamma \vdash A\ \mathit{OR}\ B{:}\mathit{BOOLEAN}}$$

$$\frac{\Gamma \vdash A{:}\mathit{INTEGER}\ \ \Gamma \vdash B{:}\mathit{INTEGER}}{\Gamma \vdash A\ +\ B{:}\mathit{INTEGER}}$$

$$\frac{}{\Gamma \vdash 0{:}\mathit{INTEGER}} \qquad \frac{}{\Gamma \vdash 1{:}\mathit{INTEGER}}$$

$$\frac{}{\Gamma \vdash \mathit{TRUE}{:}\mathit{BOOLEAN}} \qquad \frac{}{\Gamma \vdash \mathit{FALSE}{:}\mathit{BOOLEAN}}$$

These assertions will be used to fill the Abstract Syntax Tree with extra information. The Typing Rules in this example were chosen to be quite strict by prohibiting constructs like $\mathit{TRUE}\ +\ \mathit{FALSE}$. If the language should support such features it would also have been possible to define the first two typing rules like this:

$$\frac{\Gamma \vdash A{:}T\ \ \Gamma \vdash B{:}T}{\Gamma \vdash A\ \mathit{OR}\ B{:}T} \qquad \frac{\Gamma \vdash A{:}T\ \ \Gamma \vdash B{:}T}{\Gamma \vdash A\ +\ B{:}T}$$

With this definition the operations + and *OR* could be used with any type provided that the two operands have the same one. After the definition of the attributes V and assertions F the attribute grammar A can now be constructed as:

$$A = (G, V, F)$$

Using this attribute grammar a parser with a type checker would construct the Abstract Syntax Trees from the two programs, as shown in Figure 4.2 and 4.3. The first program will be parsed and checked without error messages because all applicable typing rules can be validated. The second program, however, will produce an error because the typing rule for the "+" production on the top requires that all operands i.e. child nodes must be of type INTEGER which is not the case.

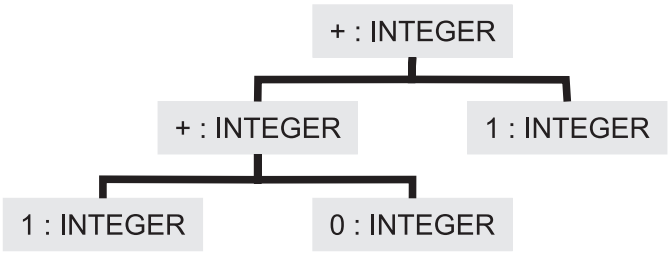

Figure 4.2: Abstract Syntax Tree Program 1

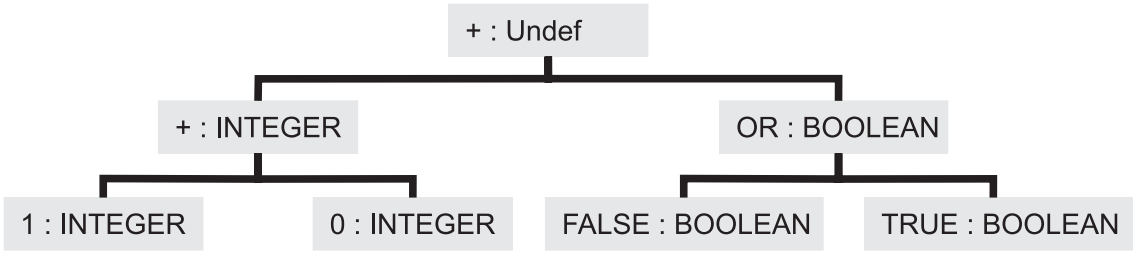

Figure 4.3: Abstract Syntax Tree Program 2

4.5 Type Checking and Type Inference

Attribute grammars make the process of type checking and type inference very elegant as the Abstract Syntax Tree with attributes can be directly mapped to a "type derivation" tree. As

demonstrated in section 3.8 this tree has to be built for every expression for a program before it can be declared as type safe. For the Wide Spectrum Type System the type checker uses defined typing rules to fill the Abstract Syntax Tree with attribute values which at the same time also builds the "type derivation" tree. Consider the following excerpt of a Strong Safe Typed WSL program and its Abstract Syntax Tree:

```
IF x = 0 THEN y := "HELLO" ELSE z := "GOOD BYE"
```

Listing 4.1: WSL Example Code

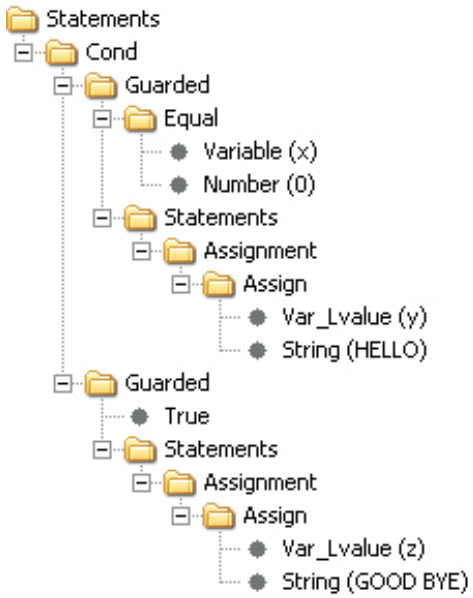

Figure 4.4: WSL Example Code

The program has the three variables x, y and z. The variable x is of type integer while y and z are strings. The process of type checking is carried out in 3 basic steps:

1. Insert the attribute values for every tree node representing a variable (Variable or Var_Lvalue).
2. Insert the attribute values for every tree node representing a static value in the program (e.g. String, Number, etc). These are mostly the typing rules which have no premise judgment.
3. Verify all expressions in the program. The type checker sequentially picks the top node of every expression and tries to verify it. In many cases this fails due to missing attribute values in one or more child nodes. In this case these child nodes (and if necessary also their child nodes) are verified first.

The typing rules have to be in a machine readable format. The rules are stored in a table with 2 columns (premise and conclusion). Table 4.1 shows an excerpt of this table and the corresponding typing rules. These typing rules are the ones which are used for the example code. The Wide

Premise	Conclusion	Typing Rule
	`Number:INTEGER`	$\frac{}{\Gamma \vdash \texttt{[0-9]+}:INTEGER}$
	`String:STRING`	$\frac{}{\Gamma \vdash \texttt{[0-9,a-z]+}:STRING}$
`Child(A):T1`	`Equal:BOOLEAN`	$\frac{\Gamma \vdash A{:}T_1 \wedge B{:}T_1}{\Gamma \vdash A = B{:}BOOLEAN}$
`Child(A):T1`	`Assign:T1`	$\frac{\Gamma \vdash A{:}T_1 \wedge B{:}T_1}{\Gamma \vdash A := B{:}T_1}$

Table 4.1: Typing Rules for Example

Spectrum Type System assesses only expressions and not whole statements because statements usually do not return any type and their correct expression has already been verified during the syntactical analysis. The first two typing rules are synthesised attributes of the attribute grammar and thus do not have any premise. They are true in every applicable situation e.g. the type INTEGER can be written straight away to the node attribute in the Abstract Syntax Tree whenever the node "Number" is encountered. The third and fourth typing rule determine inherited attributes of the attribute grammar. Thus they are based on other attributes and have a premise judgment which must be true before they can be used. The third rule can be read like: Every node "Equal" in the Abstract Syntax Tree is to be typed as BOOLEAN. All child nodes (`Child(A)`) must be of the same type (`T1`) - `T1` is therefore just a place holder for any type. The fourth typing rule is almost the same as the third except that the type of the "Assign" node is to be of the same type (`T1`) as all child nodes. Figure 4.5 shows the type check of the example code in listing 4.1. At first the synthesised attribute values for nodes which represent constants or variables are inserted. Using these attributes as a foundation, the type checker is able to determine the values of inherited attributes using the typing rules of table 4.1. If the information of variable types are missing it is possible to use these typing rules also for type inference. In this case the attributes of the nodes which represent variables become also inherited attributes. The value of these attributes is always inherited from the attribute values of their parent node. Figure 4.6 shows the type check of the example code in listing 4.1. In this way type inference is very similar to type checking. The only difference is that the attribute from nodes which represent variables are inferred from their parent

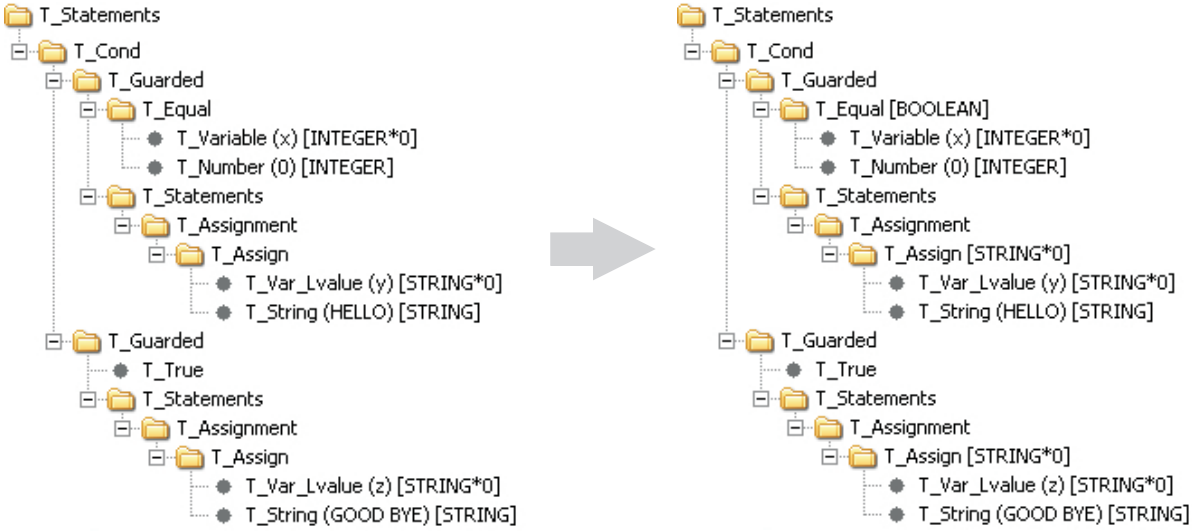

Figure 4.5: Type Check Example

node. If the parent node has no type than the attribute temporarily has the value `void`. If the variable type can be inferred somewhere else in the program the `void` attribute is set to the correct value according to the inferred type. The program is accepted if the types of all variables can

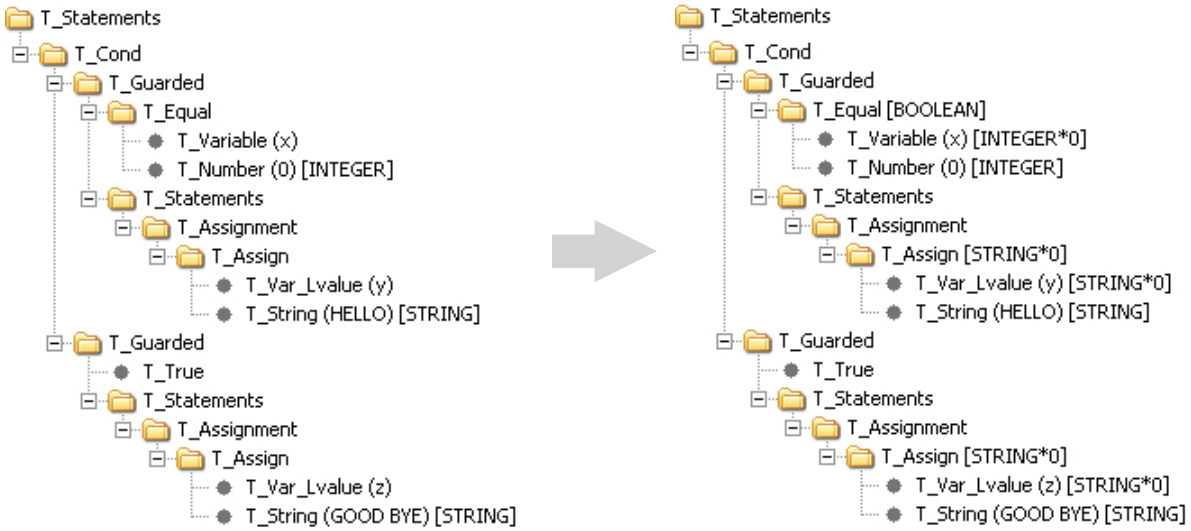

Figure 4.6: Type Inference Example

be inferred consistently as a specific type. However, the program is rejected if a variable cannot be inferred or if the inferred types of a variable differ e.g. in one place the variable is inferred as INTEGER and in another place the variable is inferred as STRING. In the Wide Spectrum Type System type inference is only used to infer the explicit typing of an untyped program or to refine explicit typing further e.g. during a type system transition when the new type system has a more

diverse type system. This means that after the type inference process is complete the inferred data types are written explicitly into the source code of the program. It is very unlikely that variables cannot be inferred as a program usually works on variables. This kind of problem can only occur if the inferred program segment is part of a bigger system and a variable is only passed through to some externally declared functions or if the variable is only used as a dummy e.g. a return value of a function which is not used. In all other cases a type can be inferred from assignments or operations. Some examples are:

- If a variable is initialised the type can be inferred from the type of the constant.
- If a variable is used in some operations the type can sometimes be inferred from the operation itself e.g. the union operator is only defined for SETs or LISTs depending on the type system layer.
- If a variable is used in some statements the type can be derived from its place in the parameter list e.g. all child nodes which represent variables of the `FOR` statement must be of type INTEGER.
- If a variable is used as parameter of a function the type can be derived if the function is declared within the processed source code or declared as `EXTERN`
- If a variable is used in an operation which involves other variables it might be possible to derive the type from one of these variables.

Although the results of type inferencing are in most cases correct and useable, it cannot replace the benefits of explicit typing as the result of type inference may not always correspond to the real intentions of the programmer. Only if the intentions of a programmer for variables are stated through explicit typing, it is possible to check and verify their consistency and validity according to the typing rules of the used type system.

4.6 Summary

This chapter gave an introduction into the Wide Spectrum Type System and its key features. Looking at the majority of commercial relevant statically typed programming languages, two

fundamental properties of typing strictness (weak or strong) and safety (safe or unsafe) can be identified. The layers of the Wide Spectrum Type System are structured accordingly so WSL can represent most of them. Special Type System Transformations can be used to move a given WSL program among these layers. Other key features are the explicit typing of the Wide Spectrum Type System, the definition of storage sizes for variables and scalability. The features of strict typing and definition of storage sizes thus have very similar purposes. Both features aim to define used data structures, regardless of the layer on which they are defined, as precisely as possible to avoid translation related errors. Introducing the Wide Spectrum Type System with attribute grammars eases the construction of typed WSL significantly as the current language definition needs only a small syntactical change to write typing expressions within the source code while finite sets of distinct attributes and assertions are just added to the definition. As long as the typing rules are correct, the set of correct programs successfully becomes a set of type correct programs. Another advantage of attribute grammars is that the type checker annotates the Abstract Syntax Tree with type derivations. The resulting type derivation trees which are directly mapped into the Abstract Syntax Tree can be easily used to reveal typing related errors. Even without explicit type declarations in the first place the typing rules and derivation trees can be used to infer data types directly from the source code with type inference.

"*The unavoidable price of reliability is simplicity.*"

Charles Antony Richard Hoare

Chapter 5

Anatomy and Realisation of the Wide Spectrum Type System

Objectives

- State general definitions and conventions.
- Provide an overview of the introduced data types.
- Discuss the layers of the type system in detail.

5.1 Introduction

This chapter explains how types and statements of the Wide Spectrum Type System are realised. It elaborates in detail which general definitions are valid for all the layers and describes all possible data types which can be found on at least one layer of the Wide Spectrum Type System. Special emphasis is put on the OSTRUCT data type which is used for object oriented structures. The chapter continues with special type directives which enable features like name alias, definition of artificial data types or external variables and with the introduction of a hierarchy of types which is used for the strongly typed layers to prevent the loss of data through type casts. The rest of the chapter goes through all layers of the Wide Spectrum Type System and explains for each layer its purpose, the supported data types and special features.

5.2 General Definitions

The purpose of a programming language is to abstract certain hardware related details away so a programmer can express more what he intends instead of how it should be realised. However, in some cases (e.g. when programming drivers for operating systems) it is inevitable and desired to define exactly how the computer should do certain hardware related operations. Therefore, every programming language has to find a balance between abstract expressions and direct hardware access. WSL is called Wide Spectrum Language because it can represent a wide range of programming constructs ranging from *low level* GOTO blocks (called Action System) to high level abstract mathematical specifications (called specification statement) [YW03]. To support the character of WSL to serve as an intermediate language during the migration process of a legacy system the Wide Spectrum Type System is an adjustable type system featuring several distinct type system layers. These layers are not designed for a particular programming language but rather for classes of programming languages in terms of their type systems. To be useful and safe, however, the Wide Spectrum Type System needs some general definitions which are the same on all of its layers. The following definitions assure that all general information which are vital for the migration process can be stated within the WSL source code:

- The definition of equality is always nominative. This means that two variables have the same type only if their declarations use the same type name.
- The type of every variable must be known.
- The size of non-container variables can be stated in bytes as part of their variable declaration with the star "`*`" declarator (e.g. `INTEGER*4` for a 4 byte INTEGER). Explicit statement of a variables precision within its declaration (see section 4.2.3) is considered to be a very important hardware related detail to prevent the unintended loss of data. The concrete data storage format, however, is not specified. Programs which need to process similar data types with a different storage[1] format may use the TYPEDEF directive (see section 5.4) to distinguish them.
- The size of container variables can be stated with a range as part of a variable declaration with

[1] An example of this would be a program which converts proprietary CRAY floating point numbers into standard IEEE 754-1985 floating point numbers.

the range "[<start>:<end>]" declarator (e.g. LIST[1:5]<INTEGER*4> a LIST containing 5 elements of 4 byte INTEGERs). It is also possible to state dynamic allocation on one or both ends of the range, either with a star (complete dynamic allocation) or with a variable (allocation upon initialisation).

- Container types can only contain content of one type. This type must always be stated in the declaration (e.g. a list which holds strings can be declared as LIST[*:*]<STRING*0>).
- Conversions between data types which are not carried out through a conversion function, have to be explicitly stated by a type cast.
- The operands of any operator must have the same type in any expression. At least one cast has to be inserted if two operands have different types.
- On all layers of the Wide Spectrum Type System, WSL offers the standard numeric operators: "+", "-", "*", "/", "**", "MOD", "DIV"; the list and string operator "++" for concatenation; the boolean operators "<", ">", "=", "<>", "<=", ">=", "IN", "NOTIN", "AND", "OR", "NOT"; and the standard SET operators: "/\","\/","\".
- Many other operations like ABS or INT are provided through intrinsic functions see [WH03] for further details.

5.3 Data Types of the Wide Spectrum Type System

This section introduces all data types which can be found on one or more layers of the Wide Spectrum Type System. Table 5.1 presents an overview of the data types which can be found on a certain layer.

5.3.1 VOID

The VOID data type is a special data type which represents the undefined type. This data type cannot be declared and is only used during the type checking and type inference process to mark a node in the Abstract Syntax Tree which has a (yet) unknown data type. An example of this is when the type inferencer encounters an assignment which involves only two variables (e.g. a := b)

Layer	VOID	POINTER	LIST	SET	SCALAR	STRING	INTEGER	BOOLEAN	HASH_TABLE	STRUCT	REAL	FIXED	COMPLEX	OSTRUCT
Dynamic Typed	√	-	(√)	-	-	(√)	(√)	-	(√)	-	-	-	-	-
Simple Typed	√	-	√	-	√	-	-	-	√	√	-	-	-	-
Weak-Unsafe Typed	√	√	√	-	-	√	√	-	√	√	√	√	-	-
Strong-Unsafe Typed	√	√	√	√	-	√	√	√	√	√	√	√	√	-
Strong-Safe Typed	√	-	√	√	-	√	√	√	√	√	√	√	√	-
OO-Typed	√	-	√	√	-	√	√	√	√	√	√	√	√	√

√ Supported; (√) Supported via implicit typing; - Not supported

Table 5.1: Overview of Data Types

during the type inference process. Both of the variables would get the type VOID because the type inferencer cannot immediately decide of which type the variables are (see section 6.3 for further details). However, once the type inferencing is finished all locations of each variable in the Abstract Syntax Tree are revisited and their inferred type must be either a single concrete type or the undefined type `VOID`. Another example are statements which are part of an expression which do not have any type (e.g. pattern variables of IFMATCH statements).

5.3.2 POINTER

The POINTER data type stores an address of computer memory. Pointers can be used in two ways:

1. As a static pointer to the memory which is set once during the declaration. After its initialisation any access to the variable would access the memory address it points to.
2. As a pseudo array which represents the memory of the computer. An access into the array would access the corresponding memory address. The pointer itself can now be represented by an ordinary integer.

In the following example the variable `val` would be filled with the content of 8 bytes in the memory starting from address $10FFA678_{hex}$:

```
EXTERN VAR <LIST[*:*]<POINTER*8> :: mem>;
```

```
VAR <
 INTEGER*4 :: ptr := 0x10FFA678,
 INTEGER*8 :: val := 0,
>:
  val := mem[ptr]
ENDVAR
```

Listing 5.1: POINTER Example

In the dynamic typed version of WSL only the 2nd version was implemented by the array `a[]` which, by default, was used as a `LIST<POINTER*1>`.

5.3.3 LIST

A LIST is an ordered collection of elements. An element in this sense can be any data type. Apart from the LIST definition of the Dynamic Typed layer all elements must be of the same type. The type of the elements must be stated when the LIST is declared. The size of a LIST is also defined during the declaration. Unlike the content type, the size parameter is optional as it differs from language to language whether the dimensions are known or dynamically managed. They can be also inserted manually if the dimensions cannot be determined by a translator directly (e.g. if the dimension is defined through `malloc` statements in C). Listing 5.2 shows the declaration of the lists `n1`, `n2` and `n3`.

```
LIST[1:5]<INTEGER> :: n1 := <1, 2, 3, 4, 5>,
LIST[*:*]<LIST[0:*]<INTEGER>> :: n2 := <<1, 2, 3>, <1, 2>, <1>>
LIST[x:y]<LIST[0:a]<INTEGER>> :: n3 := <<1, 2, 3>, <1, 2>, <1>>
```

Listing 5.2: LIST Example

The list `n1` contains 5 elements which can be addressed through `n1[1]` to `n1[5]`. Lists `n2` and `n3` are two-dimensional lists which have no explicit size. The dimension is not specified or specified during runtime with a variable. This means that the runtime environment has to take care of allocation and management. A `*` at the start index will initialise the array with a default start index (dependent on the target programming language of the migration). A possible use for this in a project is when the source language does dynamic array allocation and the array is only used with “push” and “pop” commands. In general, however, the use of “*” is dangerous and should

be avoided if possible as the resulting code is usually ambiguous and hardly portable.

5.3.4 SET

SETs are like LISTs except that they are ordered and each of its members is unique. Before introduction of a SET data type WSL uses the LIST data type as internal data structures to represent sets. Operations on a set are carried out under the assumption that it is indeed an ordered LIST where every element is unique. A list with more than one element has always to be converted to a set via the `@Make_Set` function before any set operations are applied. Only the Strongly Typed layers of the Wide Spectrum Type System contain the data type SET which is able to guarantee that set operations are always used in conjunction with SET variables.

5.3.5 SCALAR

Historically the "scalar" in computing was intended as an opposite of vector, so as to distinguish from the idea of vector processing in computer processor design. Within the Wide Spectrum Type System a SCALAR is a type of variables that can only hold single values but does not distinguish between numbers or characters. The data type SCALAR is only used for the Simple Typed WSL and its variables can represent any scalar value. Depending on their values throughout a program, the data type of all SCALAR variables has to be redefined to either INTEGER, STRING, REAL, FIXED or POINTER.

5.3.6 STRING

A string is used to store a sequence of characters which usually represent a text. Each character has the size of 1 byte encoded according to ASCII character set. Multiple STRINGs can be concatenated with the `++` operator. Unlike the C string it is not possible to access single characters of the string directly. All access to single characters has to be done through the intrinsic function `SUBSTR` which guarantees that only memory within the string variable can be addressed.

5.3.7 INTEGER

The data type INTEGER approximates the mathematical set of integers ($\mathbb{Z}$). It represents by default a signed integer in 2's complement format. However, if the definition of another representation or interpretation is needed the maintainer can use the TYPEDEF construct. Artificial data types can be used to model special interpretations like unsigned, 1's complement or BCD (Binary Coded Decimal).

5.3.8 BOOLEAN

The logical data type BOOLEAN, is a primitive data type which can have only one of two values: TRUE or FALSE. Within the computer memory it is represented as a 1 byte INTEGER which holds either 0 or 1. This data type can be used to hold any conditional expression using one or more Boolean operations such as "<", ">", "=", "<>", "<=", ">=", "IN", "NOTIN", "AND", "OR", "NOT" which correspond to some common operations of Boolean algebra and arithmetic.

5.3.9 HASH_TABLE

The hash table data structure associates keys with values. Its primary operation is to lookup a given a key and find the corresponding value. It works by transforming the key using a hash function into a so called "hash" (a number) which is used as an index in an array to locate the desired location ("bucket") where the desired value should be. HASH_TABLEs are very efficient and can insert and retrieve values usually almost instantly. A hash table may be created by setting a variable equal to HASH_TABLE. The output of example 5.3 would be "5" and "spong".

```
T:=HASH_TABLE;
T.("foo"):=5;
T.(67):="spong";
PRINT(T.("foo"));
PRINT(T.(67))
```

Listing 5.3: HASH_TABLE Example

5.3.10 STRUCT

The struct data type groups multiple variables together in a named group:

```
STRUCT :: ADDRESS := <
  SCALAR*30 :: name := "Default",
  SCALAR*20 :: line1 := "Default",
  SCALAR*20 :: line2 := "Default",
  SCALAR*20 :: city := "Default",
  STRUCT :: postcode := "Default"
    STRUCT :: outcode := <
      SCALAR*2 :: area := "XX",
      SCALAR*2 :: district := "XX"
    >,
    STRUCT :: incode := "Default"
      SCALAR*1 :: sector := "X"
      SCALAR*2 :: unit := "XX"
    >
  >
>,
ADDRESS :: strl := {ADDRESS} <
  "STRL, De Montfort University",
  "Gateway House", "The Gateway",
  "Leicester", "LE",1, 9, "BH" >,
ADDRESS :: def := {ADDRESS} < >
```

Listing 5.4: STRUCT Example

The group in Listing 5.4 declares a template. With this template it is possible to declare variables of type “Address” which will have the given structure. An initialisation can be given as a casted list of elements. The default values from the template are taken for every missing element. This ensures that every variable of the group must have an initialisation. In Listing 5.4 the variable `strl` is directly initialised with an address while the variable `def` would get the default values of the template.

5.3.11 REAL

The data type REAL is to approximate fractional numbers. It is defined as standard floating point number according to IEEE 754-1985 Standard [IEE85]. The advantage of floating-point representation over fixed-point representation is that it can support a much wider range of values. The drawback is that the numbers can be inaccurate due to normalisation and rounding errors.

5.3.12 FIXED

As the REAL data type also the FIXED data type is to represent fractional numbers. But in contrast to floating point numbers this type represents fixed point numbers. The advantage of this is that calculations with fixed point numbers are always accurate (to the specified limit of fraction digits). However, such precision limits the range of values significantly. The number has a defined number of fraction digits while all previous numbers are treated as magnitude digits. The number of fraction digits is denoted by the initialisation value (e.g. FIXED*4 :: f = 1.90F would declare a 4 byte fixed point with 2 decimal digits of fraction). As with the type INTEGER the TYPEDEF construct can be used to model special interpretations like binary fixed point numbers or BCD.

5.3.13 COMPLEX

The data type COMPLEX is to approximate the mathematical set of complex numbers. It is implemented using two REAL variables for the real and imaginary unit of the number. Complex Numbers can either be written in component ($z = x + yi$) or phasor ($z = \mathsf{z} \cdot e^{i\theta}$) form.

```
COMPLEX*4 :: c1 := CN:543.456+4.3456i,
COMPLEX*4 :: c2 := CN:543.456e4.3456p
```

Listing 5.5: COMPLEX Example

Example 5.5 shows the declaration of two complex numbers which use two 4 byte REALs. C1 is declared in component form while C2 is declared in phasor form. WSL supports all standard operations like addition, subtraction, multiplication and division.

5.3.14 OSTRUCT

The Object Oriented Typed layer introduces the OSTRUCT construct which is an enhanced version of the container data type STRUCT. It is designed to model a class template for a stateful object. Additionally to the abilities of a STRUCT (to combine several variables), an OSTRUCT can define procedure pointers and is able to inherit all definitions from other OSTRUCTS. A drawback of using procedure pointers is that all procedure names have to be unique throughout a program. However, this can be solved by using name mangling techniques like the ones used in early C++ compilers which were implemented as simple C code translators. The advantage of procedure pointers is that procedural structures are preserved while global variables which are only used by procedures within the same OSTRUCT can be consecutively localised via transformations (see section 6.5). Another advantage is the ability to support multi-inheritance. Listing 5.6 presents a very simple program written in Object Oriented Typed WSL. Modifier constructs for encapsulation like PUBLIC or PRIVATE can be used to expose procedures which are involved in dependencies between objects or to hide procedures which are only used for internal processing purposes.

```
BEGIN
  VAR <
    INTEGER*4 :: a := 0,

    OSTRUCT :: superclass := <
      PUBLIC INTEGER*4 :: b := 0,
    >,

    OSTRUCT :: class1 := <
      INHERIT superclass,

      PUBLIC INTEGER*4 :: c := 0,
      PUBLIC PROC foo(INTEGER*4)
    >,
    class1 :: object1 := < >,
  >:
    a:=1;
    object1.b = 2;
    !i object1.foo(a);
  ENDVAR
WHERE
```

```
  PROC foo(INTEGER*4 d) ==
    a := a + b + c + d
  END
END
```

Listing 5.6: OSTRUCT Example (1)

If a procedure is moved into an OSTRUCT all of its calls have to be exchanged by a corresponding call to a procedure pointer using the invoke command `!i`. Technically, the invoke command adds substitution variables for parameter and wraps a VAR environment with two parallel assignments involving the attributes of all involved objects and their current values around the call. In this exampe the call:

```
!i object1.foo(a);
```

can be mapped to:

```
VAR <b := object1.b, c := object1.c>:
  foo(p_1)
  <object1.b = b, object1.c = c>
ENDVAR;
```

With the wrapped VAR environment the OSTRUCT variables b and c become “global variables” for the procedure `foo`. The mapping between the invoke command and the wrapping VAR environment is possible because local variables are in WSL generally dynamically bound rather than statically bound (see the WSL manual [WH03]). To find the binding of a global variable in a procedure body, the interpreter looks firstly at the environment of the proc call and not the environment of the proc definition itself. In the case of this example procedure `foo` would work on the correct local attribute variables b and c. The complete code from listing 5.6 would look in Strong Safe Typed WSL like this:

```
BEGIN
  VAR <
    INTEGER*4 :: a := 0,

    STRUCT :: superclass := <
      INTEGER*4 :: b := 0,
```

```
    >,

    STRUCT :: class1 := <
      INTEGER*4 :: b := 0,
      INTEGER*4 :: c := 0,
    >,
    class1 :: object1 := < >,
  >:
    a:=1;
    object1.b = 2;

    p_1 := a;
    VAR <b := object1.b, c := object1.c>:
      foo(p_1)
      <object1.b = b, object1.c = c>
    ENDVAR;
    a := p_1

  ENDVAR
WHERE
  PROC foo(INTEGER*4 d) ==
    a := a + b + c + d
  END
END
```

Listing 5.7: OSTRUCT Example (2)

The expression of the object oriented example in a procedural layer of the Wide Spectrum Type System demonstrates the possibility to introduce object oriented features with ordinary syntactic code transformations. If a call to another object would occur inside the procedure `foo` object1 (e.g. to procedure `bar`) the object variables would need to be updated before and after the call by a parallel assignment:

```
    ...
    STRUCT :: class2 := <
      INTEGER*4 :: d := 0,
      INTEGER*4 :: e := 0,
      PUBLIC PROC bar(INTEGER*4)
    >,
    class2 :: object2 := < >,
    ...
```

```
PROC foo(INTEGER*4 d) ==
...
p_1 := b;
<object1.b = b, object1.c = c>;
VAR <d := object2.d, e := object2.e>:
  bar(p_1)
  <object2.d = d, object2.e = e>
ENDVAR;
<object1.b = b, object1.c = c>;
b := p_1;
...
```

Listing 5.8: OSTRUCT Example (3)

5.4 Directives of the Wide Spectrum Type System

To enhance the expressiveness, the Wide Spectrum Type System supports 3 directives:

- **TYPEDEF**

  The TYPEDEF directive can be used to create own specific artificial data types which are in fact sub-types of intrinsic data types. The new data type accepts the same range of values as the original data type. Technically this directive creates a child node in the hierarchy of types which guarantees that the artificial data type can always be upcasted into the intrinsic data type. The technique is especially helpful if the type system of a program has to be changed into a more expressive one. This is the case, for example, if a program from the Simple Typed layer has to be transformed to satisfy the requirements for the Weak Unsafe Typed layer. The TYPEDEF directive is used to specify the types INTEGER, STRING and POINTER. These types are then used to successively replace any SCALAR declaration according to type inference results (see section 6.5).

- **EXTERN**

  A variable can be declared as EXTERN if the variable is used in the program but declared somewhere else. With this directive it is possible to define just the type of a variable to satisfy the requirement that the type of every variable must be known.

- **SYMBOL**

  The directive SYMBOL introduces an alias for a particular variable. After this directive the variable can either be called by its original name or its alias. This directive is only available within the Simple Typed Layer and the Weak Unsafe Typed Layer and used to model aliasing constructs (e.g. in C the `#define` construct). If a program has to be transformed onto a strongly typed layer, all occurrences of the alias have to be replaced by the original name.

5.5 Hierarchy of Types

All types of the Wide Spectrum Type System can be grouped into a type hierarchy whereby each type can have several sub-types but only one super type. The sub-type (super type) relation is transitive: if C is a sub-type of B and B is a sub-type of A, then C is a sub-type of A. At the top

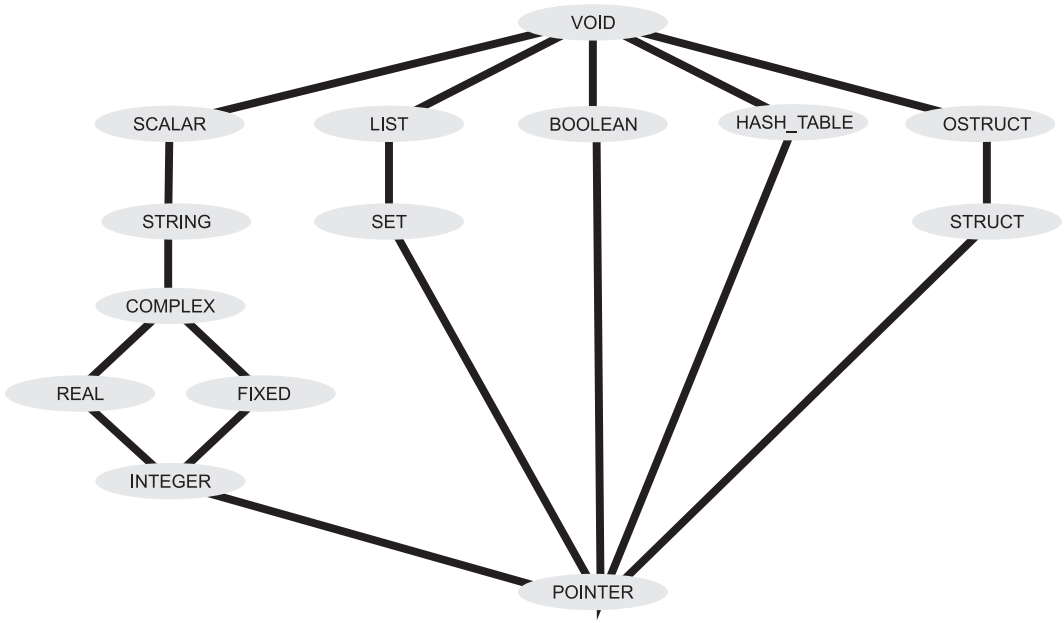

Figure 5.1: Hierarchy of Data Types

are the more general data types while the more specific data types can be found at the bottom. The most general data type which is at the top of the hierarchy is by definition the type VOID. The type VOID cannot be written directly in the source code and is only used during the type inference process for the denotation of an “unknown” type. The generality of a data type refers to its possible values. The data type POINTER is placed at the very bottom of this hierarchy as it can be casted into every other type but no other data type can be casted into a POINTER.

The set of a sub-type's possible values is always a sub-set of its super type's set of possible values. Newly defined types:

- by TYPEDEF are always direct successors of the derived intrinsic type.
- by STRUCT are always direct successors of the STRUCT type itself.
- by OSTRUCT are either direct successors of the OSTRUCT type itself or successors of the OSTRUCT types which are stated by the INHERIT directive.

The hierarchy of types is used in the strongly typed layers of the Wide Spectrum Type System to determine whether a cast is legal or not. If a type checker can verify that every type cast is an upcast (a cast from any type to one of its supertypes), it can guarantee, that no type cast causes loss of data. Examples for an upcast are `INTEGER` → `COMPLEX` or `COMPLEX` → `STRING`. The cast `REAL` → `INTEGER`, however, would be illegal as it is a downcast which might lose information.

5.6 Dynamic Typed WSL

Supported Data Types: INTEGER, LIST, STRING, HASH_TABLE[2]
Typing Appearance: Implicit
Safety: Unsafe
Type checking: Weak
Casting: Upcasting, Downcasting
Resembling Languages: Assembler

The lowest layer of the Wide Spectrum Type System is the current non-explicit typed definition of WSL. This version of WSL supports the following implicit data types: INTEGERS, LISTS/ARRAY, STRINGS and HASH_TABLE. Both, the LIST and the ARRAY type model a sequence of data in WSL. The difference between them is the implementation which has different time efficiency depending on the usage. Despite the claim of the author to also support the BOOLEAN data type, this is only true for return values i.e. a BOOLEAN value cannot be stored in a variable but the

[2]Information taken from the WSL Programmer's Reference Manual [WH03]

BOOLEAN return of a function can be used as a condition in an IF statement. The concrete definition of the data types are provided by the underlying environment which executes WSL. In the current release of FermaT this environment is provided by Scheme which uses dynamic typing. The design of the language is mainly focused on the representation of code rather than the correct representation of data. The absence of a sophisticated defined type system in this version of WSL creates several drawbacks:

- Information like length and type of a variable are almost completely missing. The only statement which involves a concrete length is the initialisation of an ARRAY. As mentioned in section 2.10 this information is stored within a separate file in its own specific intermediate data format.
- Though WSL implements operators for SETs, they only operate on ordinary LISTs. The function `@Make_Set()` just orders the elements of a LIST variable and deletes multiple entries to make sure it can be used for SET operations. If a programmer forgets to use this function prior set operations or accidently introduces a double entry again after calling `@Make_Set()`, the program may produce unexpected results in some situations. A designed SET data type, as introduced in the strongly typed layers, can prevent such situations effectively.
- The representation of memory pointers is only possible with the introduction of the pseudo array a[]. The array "a" represents thereby the memory of the computer as an array of bytes. Each element of a[] is a single byte. The notation a[n..m] represents an "array subsegment"; a list of array elements a[n], a[n+1], ..., a[m] (the list has m-n+1 elements). The notation a[n,m] represents an "array relative segment"; a list of array elements ar[n], ar[n+1], ar[n+m-1] (the list has m elements). This "workaround" was necessary as WSL does not support pointers or an extensible type system in its original language definition. Therefore, the meaning and interpretation of "a" fully depends on the individual case.
- Structures of data (e.g. `date.month := 4; date.year := 2008`) can be used but not declared in the program. The declaration of such structures is stored in the separate data structure file.
- Hex numbers are represented in Strings. This is a "workaround" to include hex numbers in WSL. This causes problems with strong typing as the result of an arithmetic operation

involving strings as operands is ambiguous.

Despite the mentioned flaws, this version of WSL was included into the Wide Spectrum Type System to be able to enhance existing programs with a sophisticated type system and to integrate the separately stored data structure information into the WSL code.

5.7 Simple Typed WSL

Supported Data Types: SCALAR, LIST, HASH_TABLE, STRUCT
Typing Appearance: Explicit
Safety: Unsafe
Type checking: Weak
Casting: Upcasting, Downcasting
Resembling Languages: Assembler

The purpose of the Simple Typed layer is to explicitly express the implicit typing of program from the untyped layer. The layer introduces the new requirement that all elements of a LIST have to be of the same type. Although, on the one hand this requirement constrains a developer, on the other hand it encourages a clean program design by means of clearly defined data structures. Conversion functions or type casts may be used if an element of a list is needed in a different type. The layer has a weak type checking. A variable can still be used as a number and as a string due to the data type SCALAR which can represent any non-container type. Furthermore, the layer introduces some features to settle several syntactic inconsistencies.

- It is now possible to transform strings containing hex numbers to standard hex numbers by writing them with "0x" as a prefix (e.g. `x := 0xA9FF`).
- Variable multireferencing (a variable can be addressed by multiple names) can be expressed through the SYMBOL directive.
- Artificial data types based on intrinsic data types can be introduced via the TYPEDEF directive.

- Variables can be grouped with the container data type STRUCT.
- Extern variables or functions can be defined with the EXTERN directive.

If a program has been successfully transferred to this layer, meaning that the program can be parsed by an appropriate parser and checked by the type checker without errors, it is guaranteed that the following requirements are satisfied:

1. Every variable is declared (either EXTERN or in the program itself).
2. The type (for this layer) of every variable is known.
3. The parameters of every function are defined.
4. Every call to a particular function must satisfy its interface.

5.8 Weak Unsafe Typed WSL

Supported Data Types: POINTER, LIST, STRING, INTEGER, HASH_TABLE, STRUCT, REAL, FIXED
Typing Appearance: Explicit
Safety: Unsafe
Type checking: Weak
Casting: Upcasting, Downcasting
Resembling Languages: C, COBOL

The diverseness of data types is extended significantly on the Weak Unsafe Typed WSL layer. The data type POINTER which can reference to any other data type is introduced. The layer resembles the type system of C most closely. Although the type system distinguishes between many data types, it does not enforce their usage. Through pointers and casts the variables can be used in any combination. This naturally gives a developer great freedom but, on the other hand, also a great deal of responsibility. Due to very limited possibilities of automated code inspection and correctness checking, it is assumed that a developer knows about all the effects and consequences

of his actions. This becomes more and more unlikely as the size of the system grows during the development process. Another drawback of casting and pointers is that expressions with extensive pointer arithmetic are extremely difficult to comprehend as the destination of a pointer reference is computed at runtime and can hardly be derived from the static source code. If a program has been successfully transferred to this layer it is guaranteed that the following requirements are satisfied:

1. The SCALAR data type from the previous layer is now split into the data types POINTER, INTEGER, REAL, FIXED and STRING. The assignment to the more specific data types is done by the type inferencer and the original source code itself.
2. The usage of pointers is clearly identified by variables which are declared as POINTER or by `LIST<POINTER>` which models the whole computer memory.

5.9 Strong Unsafe Typed WSL

Supported Data Types: POINTER, LIST, SET, STRING, INTEGER, BOOLEAN, HASH_TABLE, STRUCT, REAL, FIXED, COMPLEX
Typing Appearance: Explicit
Safety: Unsafe
Type checking: Strong
Casting: Upcasting
Resembling Languages: C, Cobol

The Strong Unsafe Typed layer introduces the three new data types SET, BOOLEAN and COMPLEX. It is no longer possible to introduce aliases via the SYMBOL directive. Each alias must now be renamed to the real variable name or declared as an own variable. Although the layer still includes the POINTER type with all its casting possibilities, it introduces constraints for casting. Only upcast according to the hierarchy of types (see section 5.5) are now allowed while every downcast has to be replaced by an explicitly written conversion function. The main aim of this layer is to strengthen the type system despite the presence of explicit pointers. This layer can be used to prepare a given legacy system for the Strong Safe Typed layer by eliminating the usage of explicit pointers step by step by accessing their target through a variable rather than through

direct memory access. This is, in fact, a very difficult and challenging task since the values of a pointer usually change during runtime which makes a statical analysis very difficult. Techniques to statically determine the possible runtime values of a pointer have a long research history and are nowadays known as pointer analysis [Hin01]. If a program has been successfully transferred to this layer it is guaranteed that the following requirements are satisfied:

1. Each variable has its own storage area. The exclusion of the SYMBOL directive prevents side effects caused by the usage of alias names.
2. No data is lost during type casts as the type check does not allow downcasting.

5.10 Strong Safe Typed WSL

Supported Data Types: LIST, SET, STRING, INTEGER, BOOLEAN, HASH_TABLE, STRUCT, REAL, FIXED, COMPLEX
Typing Appearance: Explicit
Safety: Safe
Type checking: Strong
Casting: Upcasting
Resembling Languages: C, Cobol, FORTRAN 77, FORTRAN 95

This layer has almost the same properties as the Strong Unsafe Typed layer except that the data type POINTER cannot be used anymore. With the exclusion of explicit pointers, it is no longer possible to directly access the memory and thus, every program within this layer can be considered to be as safe typed (see section 2.14.3). The main aim of this layer is to provide a possibility to represent safe typed programs. Programs which can be represented within this layer are guaranteed to be free of any pointers. Every memory access is now checked and verified as safe by the type system.

5.11 Object Oriented Typed Layer

Supported Data Types: LIST, SET, STRING, INTEGER, BOOLEAN, HASH_TABLE, STRUCT, REAL, FIXED, COMPLEX, OSTRUCT

Typing Appearance: Explicit

Safety: Safe

Type checking: Strong

Casting: Upcasting

Target Languages: Java, C#, C++, FORTRAN 95

Emerged from Simula in the 1960's the object oriented programming approach was revolutionary and remarkable successful. Although, there is still no consensus about what exactly defines an object oriented approach, certain distinctions can be identified from procedural programming languages when examining the commercially relevant approaches like Java or C#:

- Procedures and functions become methods and the modules which used to group these constructs become now stateful objects.
- A stronger coupling between data and code is encouraged by joining them in classes and objects.
- Encapsulation helps to develop clean interfaces and reduce side-effects.
- Specific concepts like composition and inheritance are introduced to increase the reusability and structure of source code.

In recent years it has become a habit for common procedural programming languages to define object oriented extensions. C/C++, FORTRAN, Perl and PHP are a few examples of languages which started as procedural languages and have been extended with object oriented facilities. The idea also to extend WSL in this way is not new and in 2007 Li defined an approach to extend WSL syntactically and semantically for object orientation [Li07]. Although, the approach adds indeed object oriented functionality, it turned out that its definition is unnecessarily complex. A careful investigation revealed that all needed facilities for object oriented features are already encoded in

the current stack of types within the procedural typed layers of the Wide Spectrum Type System. The Object Oriented Typed layer introduces the OSTRUCT data type together with the invoke statement `i!` which is needed to use the procedure pointers of OSTRUCTs. Section 5.3.14 gives a more detailed explanation of the constructs and shows that all the functionality of OSTRUCTs and the invoke commands can be completely rewritten by just using the STRUCT data type and VAR environments. This in fact means that, in contrast to Li's approach which introduced a completely new semantic relation, the approach of the Wide Spectrum Type System is able to implement object oriented structures by means of introducing new syntactical constructs alone.

5.12 Summary

This chapter presented the concrete types and statements of the Wide Spectrum Type System. It explained general definitions, the possible data types with their hierarchy, special directives for altering the type system as well as the single layers of the type system in terms of purpose, supported data types, type checking strategy as well as special features. Within the type definition the only unusual data types is the OSTRUCT. This data type is only used on the Object Oriented typed layer and is an enhanced version of the STRUCT data type which adds the possibility to define procedure pointers. Together with the invoke command `!i` it adds object oriented functionality without adding special semantics for object orientation.

"You could say that I was too lazy to calculate and so I invented the computer."

Konrad Zuse

Chapter 6

Derivation, Verification and Transformation

Objectives

- Explain the requirements and implementation of the type checking algorithm in detail.
- Show how the type checking algorithm can also be used for type inference.
- Present an algorithm for object identification in procedural based programs.
- Discuss the efficiency and validity of the algorithms.
- Present the type transformations with examples.

6.1 Introduction

This chapter presents the main algorithms which are most essential for the discussed research and explains certain important details. It begins with the algorithm which verifies type correctness for any given typed WSL program and describes how the same algorithm can also be used to infer types for untyped WSL programs. It is also shown how object structures can be identified in a procedural based program using the number of outgoing calls and a maximal allowed call depth.

The chapter concludes by listing all developed type transformations with an explanation and a small example.

6.2 Type Checking

A type system is useless without a proper type checker. In fact the type checking is as critical as the definition of the type system itself. Section 4.5 discussed and demonstrated the basic principle of type checking with an example program.

The following section will discuss the implementation of the type checking algorithm in detail. The type checker of the Wide Spectrum Type System has to satisfy the following requirements:

- The type checker must be able to adjust its level of strictness due to the layered type system approach.
- The range of allowed data types varies from layer to layer. Types have to be added or removed during every transition between layers.
- The type checks of the type checker have to be as quick and simple as possible because industrial legacy systems can easily exceed many thousand lines of code.
- The explanation of typing errors must be short and precise to ease the tracing and comprehension.

The algorithm itself is implemented in Java and needs only the typed WSL source code in form of its Abstract Syntax Tree and the typing rules in form of a table. The typing rule tables for each layer of the Wide Spectrum Type System are implemented as a hash map whose keys are conclusion nodes while the value or bucket is a list of possible applicable typing rules. These tables were constructed by listing all expression related BNF symbols from the WSL BNF definition, writing a conclusion judgement for each of them and creating premise judgments only if the type of the symbol depends on other nodes (parent or children). Table 6.1 shows an excerpt of the typing rule table for Strong Safe Typed WSL and how each typing rule would look if it were written formally.

Key	Premise	Conclusion	Formal written typing rule
`T_Number`		`T_Number:INTEGER`	$\overline{\Gamma \vdash \texttt{[0-9]+}:INTEGER}$
`T_String`		`T_String:STRING`	$\overline{\Gamma \vdash \texttt{[0-9,a-z]+}:STRING}$
`T_Equal`	`Child(A):T1`	`T_Equal:BOOLEAN`	$\frac{\Gamma \vdash A{:}T_1 \ \wedge \ B{:}T_1}{\Gamma \vdash A = B{:}BOOLEAN}$
`T_Plus`	`Child(A):T1`	`T_Plus:T1`	$\frac{\Gamma \vdash A{:}T_1 \ \wedge \ B{:}T_1}{\Gamma \vdash A + B{:}T_1}$
`T_Expressions`	`Parent=T_Sequence;` `Child(A):T1`	`T_Expressions:T1`	$\frac{\Gamma \vdash A,B,\ldots{:}T1}{\Gamma \vdash <A,B,\ldots>{:}T1}$

Table 6.1: Excerpt of Typing Rule Table for Strong Safe Typed WSL

As mentioned before in section 3.5 the conclusion statement must match an evaluated expression before the typing rule is applicable. With every applicable typing rule the conclusion judgement must hold if all premise judgments are true [Car04]. In the Wide Spectrum Type System the premise judgments can contain the following constructs:

- To identify the child nodes it is possible to write `Child(<Number>)` for a specific child. The numbering of children always starts with 0 (e.g. `Child(0)`,`Child(1)`, etc.).
- In case all children must be of the same type, it is possible to write `Child(A)` for all children.
- To identify parent nodes it is sufficient to write `Parent` as each node in the Abstract Syntax Tree can only have one parent.
- After the child or parent has been identified its type can be stored in a type variable (e.g. `T1`) with the `:` operator. This variable can be used later in the conclusion judgement to type the evaluated symbol.
- The identification of parent and children can also be used to constrain the applicability of a typing rule. If a typing rule is only applicable on nodes which have a certain child or parent it is possible to state the node names directly with the `=` operator.

The conclusion judgement assigns a type to the evaluated symbol with a type variable or with a specific value. In either way, if an evaluated node has already a specific type which is not `VOID` or the conclusion type, a type error has to be announced. For every expression which is found in a checked program there must be one typing rule whose conclusion judgement is applicable and whose premise judgment is true. The type checker will conclude a typing error if no such typing rule can be found or the conclusion statement of the first rule found cannot be verified. On a single program $\$P$ the actual type check / type inference algorithm processes as follows:

```
Write the correct type into the attribute of all nodes in the Abstract
    Syntax Tree which represent a variable.

FOR each variable $V in $P DO
  Travel through the Abstract Syntax Tree of $P and generate a list of
      nodes $N which are the top nodes of an expression where $V is
      involved.

  FOR every node $X in $N DO
    IF CALL inferLocation($X) = type error THEN RETURN type error
  OD
OD

PROC inferLocation($X)

  IF the $X has already a concrete type (not VOID) OR was visited
      before THEN RETURN the type.

  Mark $X as visited.

  Get a list of rules $L which are applicable (conclusion judgment is
      for this node type) for $X.

  IF $L is empty THEN RETURN an error.

  FOR every rule $R in $L DO
    IF no premise judgment is given THEN RETURN the conclusion type for
        $X directly.

    IF parent or one child must be a specific node type AND parent or
        child are NOT of this node type THEN RETURN type error.
```

```
    IF the premise judgment depends on child nodes AND handleChildren(
       $X,$R) fails RETURN type error.

    IF the premise judgment depends on the parent node AND handleParent
       ($X,$R) fails RETURN type error.

    $T = Apply conclusion judgment

    Check if a child can be inferred more precisely.
  OD

  CALL inferLocation(parent($X)) but do nothing if it returns a type
     error

  Check if parent can be inferred more precisely.

  IF type of $X is still not inferred RETURN with type error.

  RETURN $T
END
```

Listing 6.1: Pseudo Code for Type Checking Algorithm

See section 4.5 for an example application of the algorithm. A rough first version of this algorithm was implemented in Java as proof of concept and for performance testing on case studies. It can be found in appendix A.1

6.3 Type Inference

The type inferencer is used initially if no explicit type system was present before or in a transition into a type system layer which has higher diversity of types (e.g. Weak Unsafe Typed WSL to Strong Unsafe Typed WSL). The type inference process uses the same algorithm as the type checking process with the exception that the types of variables are not written to the nodes in the Abstract Syntax Tree before the algorithm starts as they are not known (see section 4.5). Another difference is that the loop of the algorithm is executed several times. Whenever a type of a variable is inferred at some location of the code its type is stored and the variable is marked as *known*. All type information in the Abstract Syntax Tree are then deleted and the new *known* type together

with all other *known* types are written to the corresponding nodes in the tree. With the new information in the tree the algorithm is executed again. The type inference finishes if the types of all variables are known or if no new variable types can be inferred. The total number of executions depends on the source code and is usually not more than three times. The user has to specify the type manually if the type of a variable cannot be inferred. Such situations do occur but they are quite rare as most variables are initialised at some point or are used according to their value. If a variable stores values of different types at different points in the program (e.g. variables which represent registers in the translation of an assembler program) the inference process would fail too. In these cases a possible solution might be to transform the program into static single assignment (SSA) form in which every variable is assigned exactly once or to use type transformations such as Separate Multi-Typed Scalars (see section 6.5).

6.4 Object Identification in Procedural Based Programs

The object oriented paradigm became very interesting for software comprehension, in recent years, as object oriented programs tend, in general, to be more structured and concise and hence more readable. With the help of current state-of-the-art object oriented modelling techniques a developer is able to create structures which model real world objects very closely, making the produced code intuitively comprehensible and easily reusable. As described in section 2.11 the discovery and development of sophisticated approaches for object identification has been the subject of many scientific investigations since the early 90s. Most of these approaches are either data centric or code centric oriented. The object identification technique of the proposed approach is a refined version of a method which was first published by Pidaparthi et. al. in 1998 [PZL98]. The authors used the view of the software life cycle, in which all software development is considered to be an evolutionary activity with re-engineering/restructuring as an important process which is applied repeatedly. However, while this approach favored putting all code and data at first in one so called "God Class" and then starting to extract smaller classes, the presented approach will already identify several classes in the first step fully automated. Unlike many data centric approaches the FermaT approach utilises for this the call graph as starting point for its object identification. After the first fully automated object identification, a maintainer is able to tweak the restructuring of the system further by applying Type System Transformations. This approach has several advantages:

- Experience has shown that the simplification based on the call graph i.e. elimination of inter-class relations in object oriented systems contribute significantly to the comprehensibility of software [Mil04].
- The Maintainer has the ability to configure the first automatic stage of the object identification process, thus having the choice among several results to use as starting point for the transformation based restructuring.
- Every migration step is small and traceable. The maintainer has therefore the ability to influence the migration process in many different ways. Especially this property has proved to be very useful and has significantly contributed to the success of FermaT in the past.

The fully automatic object identification algorithm for the first steps has four parameters:

- **Fan-Out Threshold** ($F) is the number of outgoing calls for a procedure to be identified as the main method of a class.
- **Class Cluster Depth** ($D) is the possible call depth within one class.
- **Class Identification Order** ($O) defines which methods are used first for class identification. These can either be the methods with the most or smallest amount of outgoing calls. The size of the single classes is highly depended on this.

The algorithm itself processes as follows (see the case studies in section 8.2 and 8.3 for an example application):

```
FOR all procedures $P DO
  IF $P has a Fan-Out >= Fan-Out Threshold $F THEN add $P to list $L of
       initial class methods.
OD

FOR all methods $M in $L (iteration according Class Identification
    Order $O) DO
  Create a new class $C with $M as first method.

  IF $M is not already part of another class THEN include all
      procedures which are not already part of another class and
```

```
      reachable within Class Cluster Depth $D steps from $M into class
      $C.

  Add $C to a list of identified classes $I.
OD

Declare all methods which are only called from within its class as
    private.

Distribute all variables which are accessed only from one class to this
     class and declare the new attribute as private.
```

Listing 6.2: Pseudo Code for Object Identification Algorithm

All classes are by default held in public variables and all references to procedures and public variables in the code are updated[1]. Once the initial object identification is completed and has been accepted by the maintainer, the object oriented system can either be directly translated to the designed target language or refined further by applying transformations. Procedures and variables which could not be distributed are put into the global scope. When migrated into an object oriented language, the global scope items become public static methods and attributes of a single class called global.

6.5 Type System Transformations

The following table lists all transformations which can be used to modify WSL code with a type system.

#	**Type System Transformation**	**Available in Layer(s)**
1	Insert Variable Declarations	Untyped WSL
2	Insert Type Annotations	Untyped WSL
3	Hex String to Number	Simple Typed WSL
4	Separate Multi-Typed Scalars	Simple Typed WSL
5	Insert Type Casts	Simple Typed WSL
6	Specify Scalar Variables	Simple Typed WSL
7	Declare Pointer variable	Simple Typed WSL
8	Rewrite SYMBOL Variables Weak Unsafe Typed WSL	Simple Typed WSL
9	Rewrite Pointer Variables for Arrays and Structures	Simple Typed WSL Weak Unsafe Typed WSL

[1] The concrete implementation of this depends on the target language.

		Strong Unsafe Typed WSL
10	Rewrite INTEGER to BOOLEAN Variables	Strong Unsafe Typed WSL
11	Rewrite LIST to SET Variables	Strong Unsafe Typed WSL
12	Localise Item	Object Oriented Typed WSL
13	Globalise Item	Object Oriented Typed WSL
14	Insert Public Procedures	Object Oriented Typed WSL
15	Localise Procedures	Object Oriented Typed WSL
16	Insert Public Variables	Object Oriented Typed WSL
17	Localise Variables	Object Oriented Typed WSL
18	Merge classes	Object Oriented Typed WSL
19	Partition class	Object Oriented Typed WSL
20	Reduce inter-class relations	Object Oriented Typed WSL
21	Create composition	Object Oriented Typed WSL
22	Extract super class	Object Oriented Typed WSL
23	Declare Inheritance	Object Oriented Typed WSL

Table 6.2: Initial Type System Transformation Bank

The correctness of a transformation result should be verified by a successful type check after every transformation or a series of transformations to minimise the risk of implementation related errors. Many transformations for the object oriented layer were inspired by previous work by Pidaparthi, Zedan and Luker [PZL98]. In the future development this bank will be modified and extended.

6.5.1 Type Transformations For Procedural Typed Layers

The following type transformations are used on the procedural typed layers of the Wide Spectrum Type System to raise the level of type consistency by refining the declaration of types and usage of variables.

Transformation Number: 1

Transformation: Insert Variable Declarations

Layers: Untyped WSL

Preconditions: One or more variables are used without being initialised.

Description: On all explicitly typed layers of the Wide Spectrum Type System it is required that every used variable is declared with its type and initialised with an initial value. This transformation searches for variables in the Untyped WSL and uses the type inferencer to infer their type. The missing variables are then declared and initialised with their first assignment or with

standard values (e.g. 0 for SCALARs, empty list for LISTs, etc.). External declared variables can be marked with the statement EXTERN.

Example:

Before transformation	After transformation

`j := 4;`
`...`
`i := j;`
	`VAR<j := 4,i := 0>:`
	`...`
	`i := j;`

Table 6.3: Example Insert Variable Declarations

Transformation Number: 2

Transformation: Insert Type Annotations

Layers: Untyped WSL

Preconditions: An untyped WSL program where all variables are initialised.

Description: This transformation executes the type inference algorithm and declares all variables according to their inferred type. The type system layer changes from Untyped to Simple Typed.

Example:

Before transformation	After transformation

`j := 4;`
`j := j + 4;`
	`INTEGER*0 :: j := 4,`
	`...`
	`j := j + 4;`

Table 6.4: Example Insert Type Annotations

Transformation Number: 3

Transformation: Hex String to Number

Layers: Simple Typed WSL

Preconditions: Variables which hold a string with a hex value are never used explicitly as string.

Description: Procedure Due to a lack of expressiveness the current version of WSL has to express numbers which are initialised in hex format as strings. This is an unclean definition and therefore

the definition of Simple Typed WSL introduces the standard hex notation 0x*number* (e.g. `0xFF76`) which makes it possible for this transformation to transform hex strings into numerical numbers.

Example:

Before transformation	After transformation
`STRING*0 :: i := "",` `i = "hex 0xFF";`	`INTEGER*0 :: i := "",` `i = 0xFF;`

Table 6.5: Example Hex String to Number

Transformation Number: 4

Transformation: Separate Multi-Typed Scalars

Layers: Simple Typed WSL

Preconditions: One or more single scalar variables are used to store different data types.

Description: As the untyped version of WSL uses the dynamic typing of the underlying SCHEME environment it might happen that a variable holds data of different data types throughout the program. In these cases it is not possible to assign a single data type to these variables. This transformation is able to solve this issue by declaring two variables of different types. The names of the variables are changed in the program according to their type usage.

Example:

Before transformation	After transformation

`SCALAR :: a := 0,`
`...`
`a := 5**10;`
`...`
`a := "Some String" ++ a;`
`PRINT(a)`
`...`
`a := 4;`
`a := a * 5;`
	`SCALAR :: i_a := 0,`
	`SCALAR :: s_a := 0,`
	`...`
	`i_a := 5**10;`
	`...`
	`s_a := s_i;`
	`s_a := "Some String" ++ s_a;`
	`PRINT(s_a);`
	`...`
	`i_a := 4;`

	`i_a := i_a * 5;`

Table 6.6: Example Separate Multi-Typed Scalars

Transformation Number: 5

Transformation: Insert Type Casts

Layers: Simple Typed WSL

Preconditions: The type check fails because variable of different (but compatible) types are used in a formula without type casts.

Description: The Wide Spectrum Type System puts emphasis on explicit typing and explicit type casts. Type casts must be inserted whenever variables of different types are used within the same formula. This transformation can insert such casts into every expression of the program. In fact the purpose of this transformation is to explicitly write every implicit type cast.

Example:

Before transformation	After transformation
`INTEGER*4 :: i := 0,`	`INTEGER*4 :: i := 0,`
`INTEGER*8 :: j := 0,`	`INTEGER*8 :: j := 0,`
`...`	`...`
`j := i;`	`j := {INTEGER*8} i;`

Table 6.7: Example Insert Type Casts

Transformation Number: 6

Transformation: Specify Scalar Variables

Layers: Simple Typed WSL

Preconditions: Every variable of the program stores data of only one type. All data types from Weak Unsafe Typed WSL can be inferred by the typed inferencer.

Description: The transformation introduces new data types with the TYPEDEF command. These new data types are the same as the new data types in Weak Unsafe Typed WSL except the POINTER data type. The transformation prepares the transition from Simple Typed WSL to

Weak Unsafe Typed WSL by retyping variables with the new data types according to their inferred type for Weak Unsafe Typed WSL.

Example:

Before transformation	After transformation

`SCALAR*4 :: i := 0,`
`SCALAR*20 :: j := "",`
	`TYPEDEF INTEGER (SCALAR);`
	`TYPEDEF INTEGER (STRING);`
	`...`
	`INTEGER*4 :: i := 0,`
	`STRING*20 :: j := "",`

Table 6.8: Example Specify Scalar Variables

Transformation Number: 7

Transformation: Declare Pointer variable

Layers: Simple Typed WSL

Preconditions: The program uses the pointer variable `a[]`.

Description: The transformation introduces the new data type POINTER with the TYPEDEF command. The transformation declares the variable `a` as `LIST[*:*]<POINTER*1>` and inserts all type casts accordingly.

Example:

Before transformation	After transformation

`LIST[*:*]<SCALAR*0> :: a :=`
`  {LIST[*:*]<SCALAR*0>} < >,`
`...`
`r11 := a[r1,4];`
	`TYPEDEF POINTER (SCALAR);`
	`...`
	`LIST[*:*]<POINTER*1> :: a :=`
	`  {LIST[*:*]<POINTER*1} < >,`
	`...`
	`r11 := {SCALAR} a[r1,4];`

Table 6.9: Example Declare Pointer variable

Transformation Number: 8

Transformation: Rewrite SYMBOL Variables

Layers: Simple Typed WSL, Weak Unsafe Typed WSL

Preconditions: The program uses the directive SYMBOL.

Description: The transformation inserts the real name of a variable in every occasion where a SYMBOL alias is used throughout the SYMBOL context.

Example:

Before transformation	After transformation

`SYMBOL j i`
`...`
`INTEGER*4 :: i := 1,`
`...`
`IF (j = 1) THEN`
`  EXIT(1) FI`
	`INTEGER*4 :: i := 1,`
	`...`
	`IF (i = 1) THEN`
	`  EXIT(1) FI`

Table 6.10: Rewrite SYMBOL Variables

Transformation Number: 9

Transformation: Rewrite Pointer Variables for Arrays and Structures

Layers: Simple Typed WSL, Weak Unsafe Typed WSL, Strong Unsafe Typed WSL

Preconditions: The program uses POINTER variables which can be rewritten through array- or structure variable access.

Description: The transformation tries to rewrite the access through pointers on array and STRUCT structures with access through variables.

Example:

Before transformation	After transformation

`LIST[*:*]<POINTER*1> :: a :=`
`  {LIST[*:*]<POINTER*1>} < >,`
`...`
	`IF WTAB1.WDS1F1[1] = 0xFF THEN`
	`  EXIT(1)`

`WDS1 := !XF address_of(WTAB1);`
`IF a[WDS1].WDS1F1[1] = 0xFF THEN`
`  EXIT(1)`	

Table 6.11: Rewrite Pointer Variables for Arrays and Structures

---

Transformation Number: 10

Transformation: Rewrite INTEGER to BOOLEAN Variables

Layers: Strong Unsafe Typed WSL

Preconditions: INTEGER variables are used only in conditions as BOOLEAN and nowhere else.

Description: Because C does not know BOOLEAN variables and uses INT variables instead it is in many cases possible to rewrite INT variables as BOOLEAN. The C construct `if (SWITCH) { break; }` for example would be translated as `IF (SWITCH <> 0) THEN EXIT(1) FI`. With the introduction of a BOOLEAN data type in the Strong Unsafe Typed WSL layer these constructs can be rewritten with this transformation.

Example:

Before transformation	After transformation

`INTEGER*4 :: SWITCH := 1,`
`...`
`IF (SWITCH <> 0) THEN`
`  EXIT(1) FI`
	`BOOLEAN*0 :: SWITCH := TRUE,`
	`...`
	`IF (SWITCH) THEN`
	`  EXIT(1) FI`

Table 6.12: Rewrite INTEGER to BOOLEAN Variables

Transformation Number: 11

Transformation: Rewrite LIST to SET Variables

Layers: Strong Unsafe Typed WSL

Preconditions: LIST variables are used as sets.

Description: In the untyped version of WSL a LIST can be "converted" to a set via the `@Make_Set` function. However, the internal data type is still a LIST and there is no mechanism which assures that the variable is not used as a LIST afterwards (e.g. by using the operator `++` instead of `\/`). This transformation searches for SET variables and declares them correctly as SETs.

Example:

Before transformation	After transformation
`LIST[1:10]<INTEGER*4> :: x := < >,`	`SET[1:10]<INTEGER*4> :: x := < >,`
`LIST[1:10]<INTEGER*4> :: y := < >,`	`SET[1:10]<INTEGER*4> :: y := < >,`
`...`	`...`
`x := @Make_Set(<1, 2, 3, 4>);`	`x := @Make_Set(<1, 2, 3, 4>);`
`y := x \/ <5>;`	`y := x \/ <5>;`

Table 6.13: Rewrite LIST Variables to SET Variables

6.5.2 Type Transformations For Object Oriented Typed Layers

The following examples are illustrated with class diagrams. The accessibility of attributes and methods are denoted with the standard annotations: + for a PUBLIC item and − for a PRIVATE. PRIVATE items can only be accessed from within the same class while PUBLIC items can be accessed from everywhere. An access from a procedure to a procedure (a call) or attribute is shown with an arrow. Many of the following transformations require user interaction. The main purpose of them is to assist in the further refinement of an object oriented system after the object identification algorithm has been applied. The usage of such transformations is much safer than

the manual modification as, for example, references of moved items[2] can be automatically updated throughout the whole program.

Transformation Number: 12

Transformation: Localise Item

Layers: OO Typed WSL

Preconditions: Some procedures or variables are not in the scope of an OSTRUCT (globally defined). **Description**: A global procedure or variable is moved into an existing or new OSTRUCT as a public attribute or method. This transformation is used to cluster either procedures or variables of a procedural system into objects.

Example:

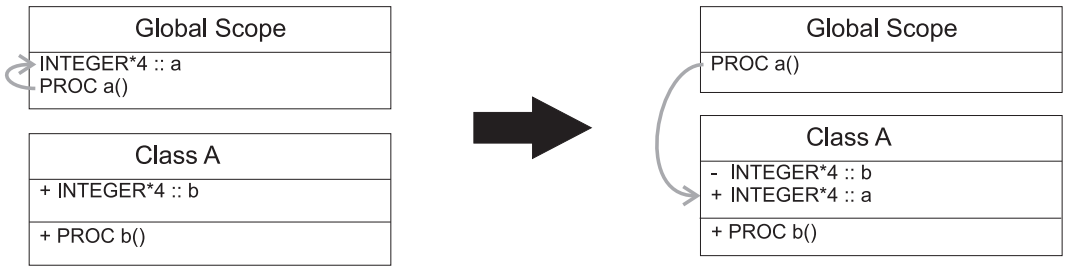

Figure 6.1: Localise Item

Transformation Number: 13

Transformation: Globalise Item

Layers: OO Typed WSL

Preconditions: Some methods or attributes are declared in the scope of an OSTRUCT.

Description: A method or attribute is moved from an existing OSTRUCT scope into the global scope which is the reverse of "Localise Item".

Example:

[2] A moved item in this context for example a public procedure which is "moved" into the scope of an OSTRUCT.

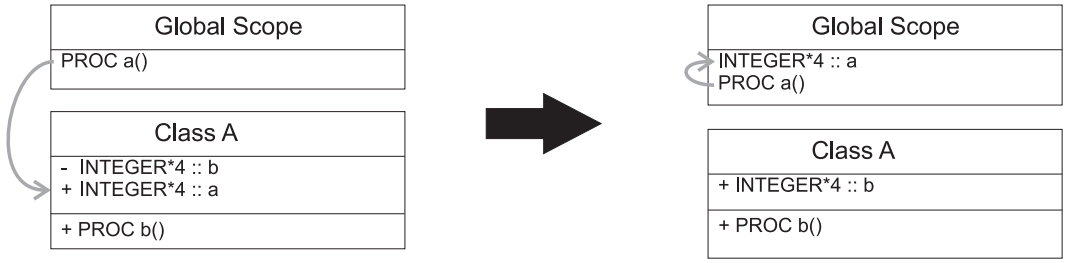

Figure 6.2: Globalise Item

Transformation Number: 14

Transformation: Insert Public Procedures

Layers: OO Typed WSL

Preconditions: Some procedures are not in the scope of an OSTRUCT (globally defined).

Description: This transformation distributes all global procedures among all existing OSTRUCTs and declares them as public. It tries to introduce as less new inter-class relations as possible. However, the introduction of new inter-class relations is sometimes necessary when for example a distributed procedure is called from methods which are defined in different classes.

Example:

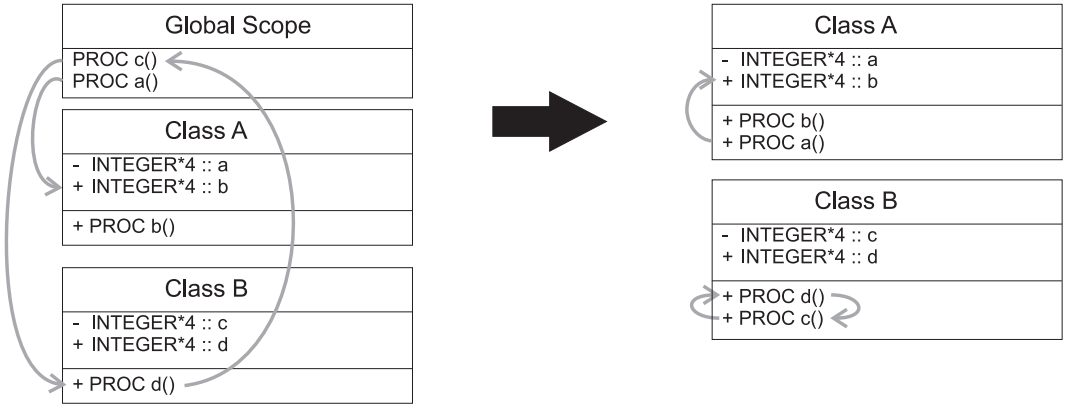

Figure 6.3: Insert Public Procedures

Transformation Number: 15

Transformation: Localise Procedures

Layers: OO Typed WSL

Preconditions: Some procedures are public inside an OSTRUCT defined or not in the scope of an OSTRUCT (globally defined). Many variables have been distributed among OSTRUCTs.

Description: This transformation tries to localise (i.e. declare as private) as many procedures as possible. Procedures which do not introduce new inter-class relations (access requests from one OSTRUCT procedure to procedures or variables in other OSTRUCTs) are moved into OSTRUCTs. Every procedure which is not involved in an inter-class relation is declare as PRIVATE. A procedure is not changed if methods of multiple OSTRUCTs or global procedures are calling it.

Example:

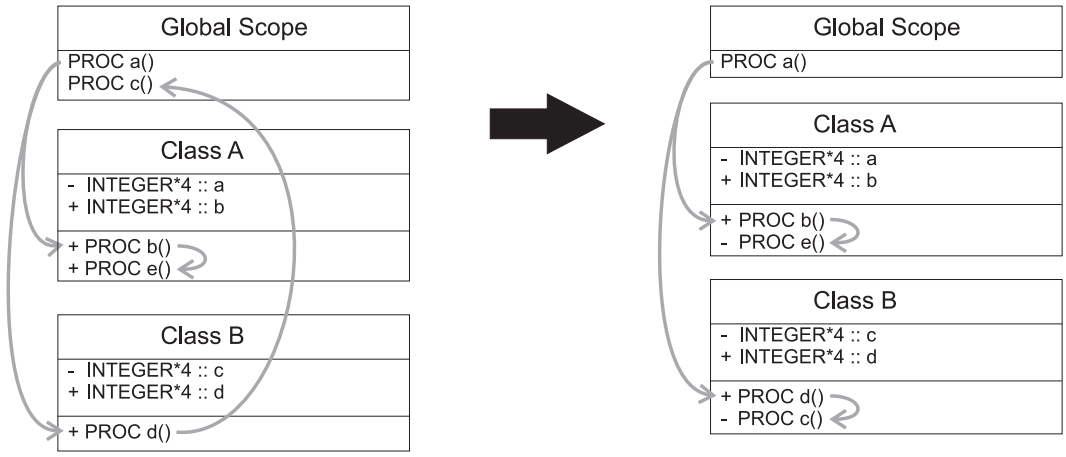

Figure 6.4: Localise Procedures

Transformation Number: 16

Transformation: Insert Public Variables

Layers: OO Typed WSL

Preconditions: Some variables are not in the scope of an OSTRUCT (globally defined).

Description: This transformation distributes all global variables among all existing OSTRUCTs and declares them as public. It tries to introduce as less new inter-class relations (access requests

from one OSTRUCT to a variable of another OSTRUCT) as possible.

Example:

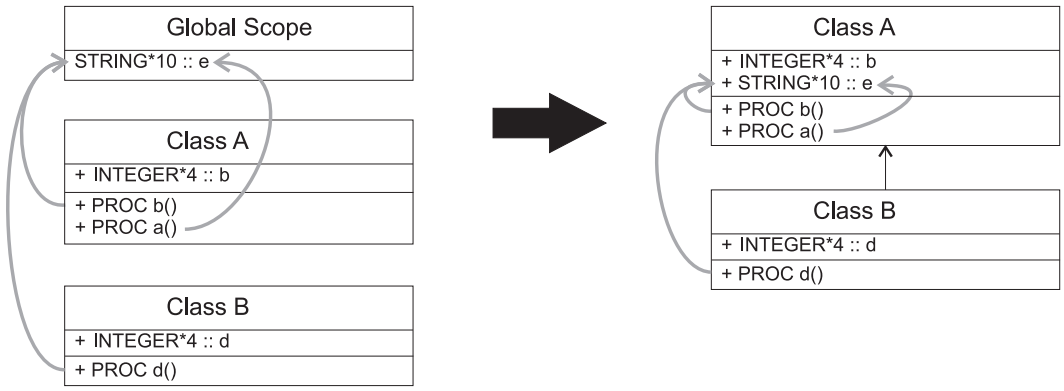

Figure 6.5: Insert Public Variables

Transformation Number: 17

Transformation: Localise Variables

Layers: OO Typed WSL

Preconditions: Some variables are public inside an OSTRUCT defined or not in the scope of an OSTRUCT (globally defined). Many procedures have been distributed among OSTRUCTs.

Description: This transformation tries to localise (i.e. declare as private) as many variables as possible. Variables from the global scope which are only accessed from one class are moved into it. Every variable in a class which only accessed by local methods is declare as PRIVATE.

Example:

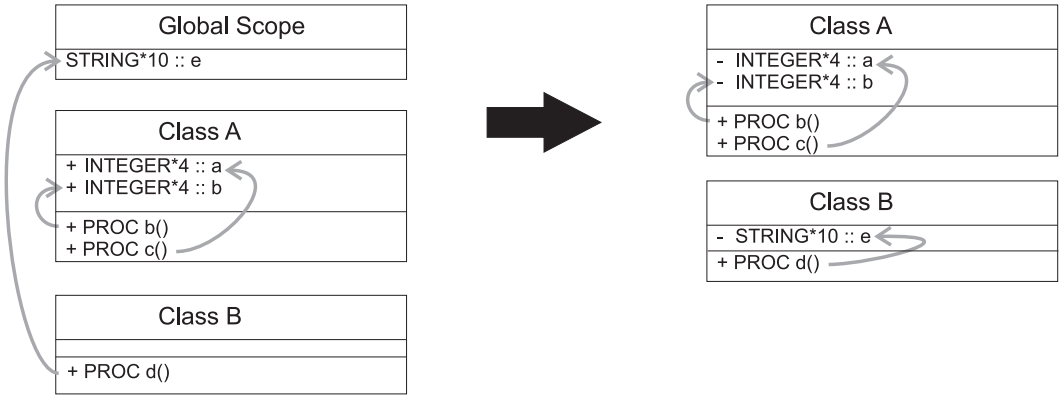

Figure 6.6: Localise Variables

Transformation Number: 18

Transformation: Merge Classes

Layers: OO Typed WSL

Preconditions: The system has at least 2 OSTRUCTs.

Description: Merges all methods and attributes of two OSTRUCTs into one OSTRUCT and updates all their references.

Example:

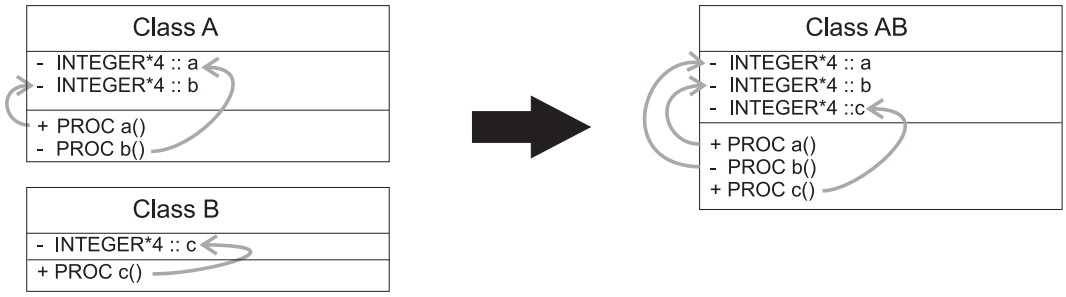

Figure 6.7: Merge Classes

Transformation Number: 19

Transformation: Partition Class

Layers: OO Typed WSL

Preconditions: The system has at least 1 OSTRUCT.

Description: Moves selected methods and attributes from one OSTRUCT into a new OSTRUCT and updates all references to them. This transformation is the reverse of "Merge classes".

Example:

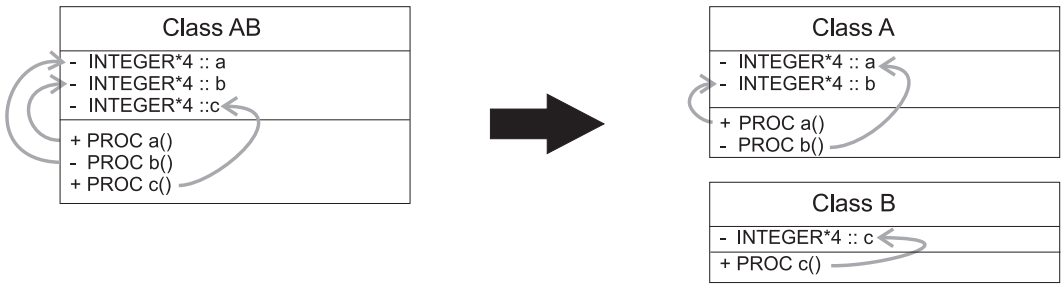

Figure 6.8: Partition Class

Transformation Number: 20

Transformation: Reduce Inter-Class Relations

Layers: OO Typed WSL

Preconditions: The system has at least 2 OSTRUCTs.

Description: Reduces the calls between two OSTRUCTs by swapping methods between them. Public methods are swapped which have more calls by methods from the other OSTRUCT than by the methods of the current one. All references to the moved methods are updated.

Example:

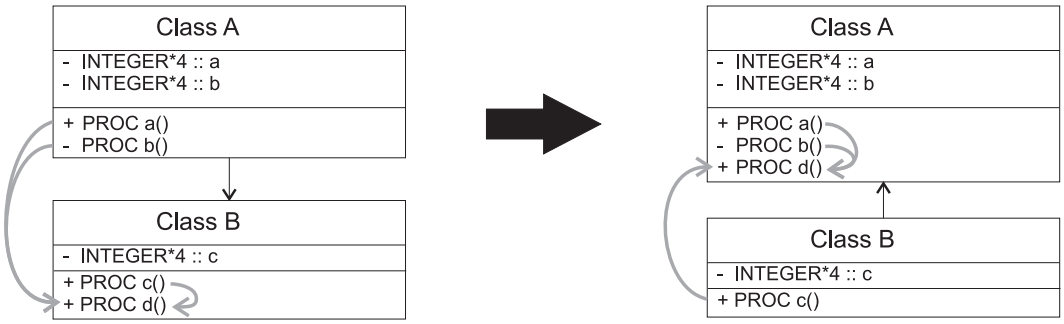

Figure 6.9: Reduce Inter-Class Relations

Transformation Number: 21

Transformation: Create composition

Layers: OO Typed WSL

Preconditions: The system has at least 2 OSTRUCTs.

Description: Puts one class into the definition of another class and updates all its references.

Example:

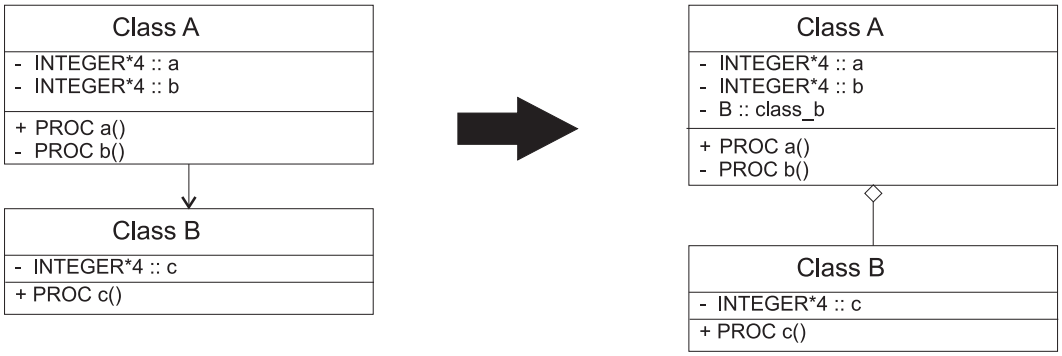

Figure 6.10: Create Composition

Transformation Number: 22

Transformation: Extract super class

Layers: OO Typed WSL

Preconditions: The system has at least one OSTRUCT.

Description: Extracts several specified items and puts them into a super class.

Example:

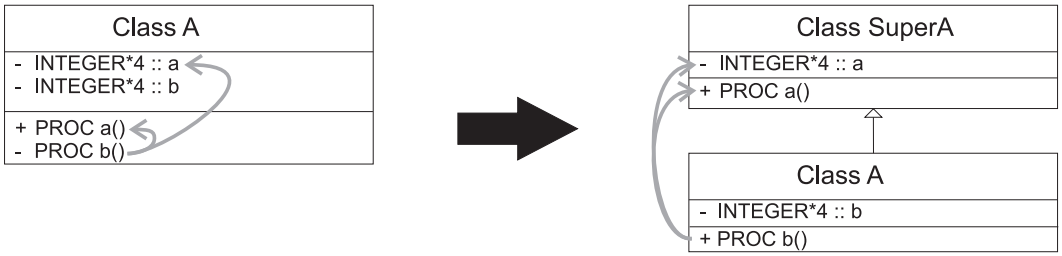

Figure 6.11: Extract Super Class

Transformation Number: 23

Transformation: Declare Inheritance

Layers: OO Typed WSL

Preconditions: The system has an inheritance hierarchy and at least one class which has no super class.

Description: Includes a class into an existing inheritance hierarchy. All methods are kept as they are while public variables which are used by both classes are declared as static. Public variables which are only used by both involved classes are declared as private.

Example:

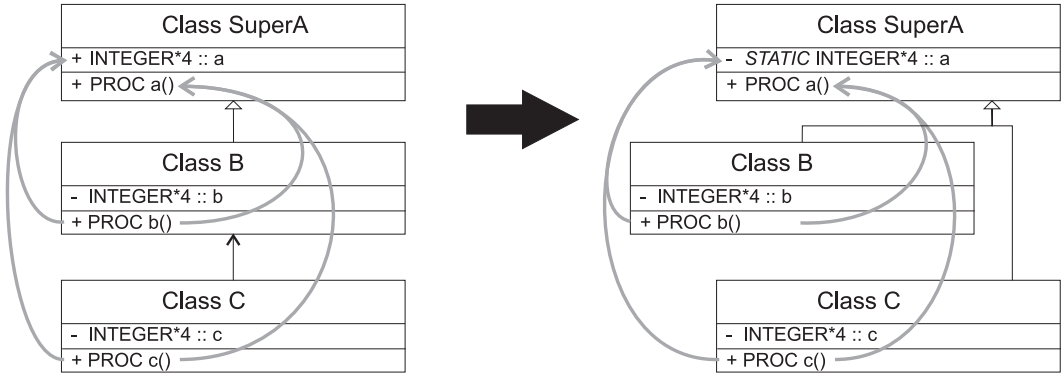

Figure 6.12: Declare Inheritance

6.6 Summary

This chapter presented the main algorithms for the Wide Spectrum Type System and elaborated certain important details. The first presented algorithm was the type checking algorithm which is able to adjust its level of strictness and range of allowed data types depending on the used typing rules. It is able to check large-scale systems within reasonable time and reports found typing errors as briefly and precisely as possible. The same algorithm can also be used to infer data types if no explicit typing information are available. The only difference between the two algorithms is that in case of the type inference algorithm, the types of variables are not written to the nodes in the Abstract Syntax Tree prior execution of the algorithm. The chapter also presented the object identification algorithm which can be used to identify object structures in a procedural

based program as well as the type system transformations which provide the facilities to move a given WSL program among the layers of the Wide Spectrum Type System and, in case of the transformations for the Object Oriented Layer, to change the architecture paradigm of a legacy system from procedural oriented to object oriented.

"*Make it look sexy!*"

Hussein Zedan

Chapter 7

Tool Support

Objectives

- To review the development history of the FermaT transformation system.
- To describe the architecture and implementation of FermaT Maintenance Environment.
- To describe the architecture and implementation of the FermaT Type System Editor.
- To propose a possible integration of this technology into the FermaT transformation system.

7.1 Introduction

This chapter reviews and describes the tools which have been developed for the implementation and support of the FermaT transformation theory. The chapter starts by giving a brief review about the development history of the FermaT transformation system. It then describes the developed programs for this research called FermaT Maintenance Environment, a graphical user interface for the FermaT transformation system which was build to understand the internal processing of the FermaT transformation system, and the FermaT Type System Editor, which is the implementation of the presented research about the Wide Spectrum Type System. The description of the two tools includes explanation of all their functions and internal data structures as well as information about their internal code structure and orgranisation. The chapter finalises by proposing an integration of a type check to the current FermaT transformation process.

7.2 Development History

The implementation started by examining the functionality of the FermaT transformation system. The free version is available for download from the Homepage of Martin Ward[1]. The only difference between the free (called fermat3) and the commercial version (called fermat2) of FermaT is the number of available transformations. The FermaT transformation system is a command line based tool which can apply transformations to a given WSL program. The tool comes in a package together with its source code written in WSL and some documentation (mainly the WSL Programmer's Reference Manual WSL [WH03]). Although, the documentation is sufficient for an experienced programmer to be able to use the transformation system, it does not explain much about the internal processes. Unfortunately, it turned out that such a documentation does not exist at all. In fact most of the necessary knowledge about the internal functions had to be collected directly through reverse engineering. The first major goal for the practical development during the

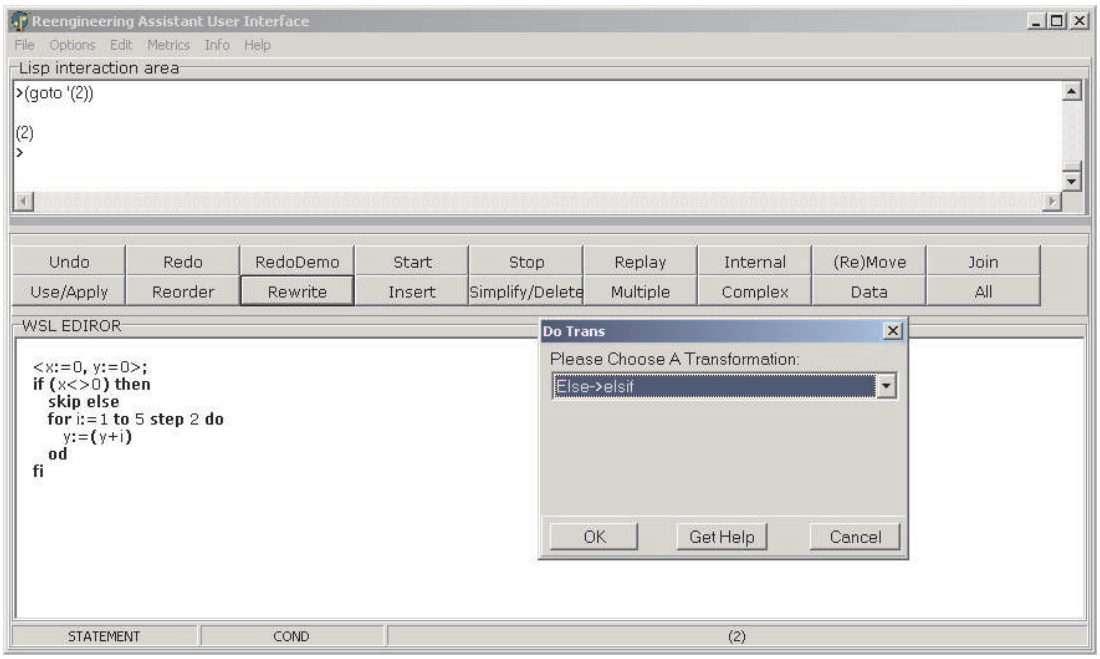

Figure 7.1: FermaT Maintainers Assistant

research investigation was to build a graphical interface to access the functionality of the tool more easily and to understand how the internal processes work. The idea of building such a graphical interface was not new. Already in the early days of FermaT in the 1990's the prototype tool Main-

[1] http://www.cse.dmu.ac.uk/∼mward/fermat.html

tainer's Assistant had a graphical user interface. However, as mentioned in section 2.6, the whole system lacked time and space efficiency. When the system was reimplemented in $\mathcal{META}$WSL the graphical user interface was abolished in favor of a pure command line based interface. This was a logical step since the development focused on archiving better performance and the automation of the transformation process. For the control scripts, which are currently steering the transformation process, a graphical user interface would be useless and only consuming system resources. However, for testing, learning and program comprehension purposes a graphical user interface is needed. Also for this research, to be successful, a general understanding of the FermaT software and its data structures was needed as well as a graphical user interface which provides a more accessible API than the current $\mathcal{META}$WSL / Scheme implementation. These considerations lead to the development of a graphical user interface called FermaT Maintenance Environment (FME).

7.3 FermaT Maintenance Environment

The first step in the implementation of the FME was to identify how the application would be used and which requirements can be specify the requirements for the such an application. The following use cases were identified for the FME: The FME's focus is to assist in the comprehension of WSL

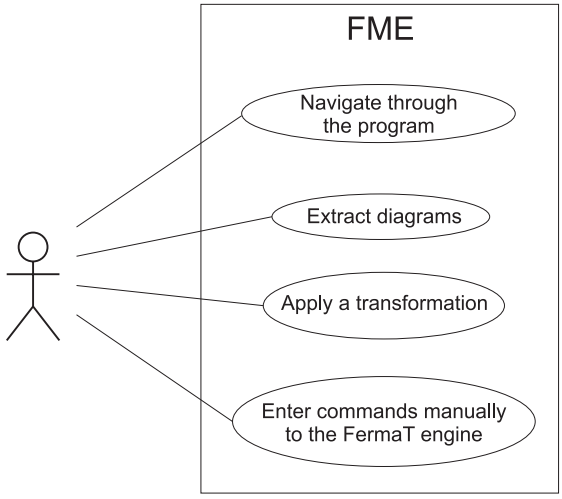

Figure 7.2: FME UML Use Case Diagram

source code and the effect of transformations. The application is not time critical as a human user acts much slower than for example a script. The goal is to build a small application which is able to represent WSL source code and to control the most important functions of the FermaT engine.

Following these considerations, the following requirements can be identified:

- Easy-to-use navigation through WSL programs via source code or Abstract Syntax Tree.
- Selected code statements are highlighted in the source code and the Abstract Syntax Tree.
- Transformations are easy accessible through a transformation catalogue. The application of a transformation occurs on the current select item in the Abstract Syntax Tree.
- Commands can be directly passed to the engine via a command console.
- Diagrams can be extracted from the source code for better comprehension.
- Multiple WSL files can be incorporated into a project.

For the FME the Java programming language was the implementation language of choice as it has a huge API with many available extension libraries, mostly developed by the open-source community. Java is a robust object oriented language with an explicit strong and safe type system. Its virtual machine concept makes it fully platform independent, allowing to run the FME together with the FermaT transformation system on Windows and most Unix based platforms. The language itself has a straight forward syntax with almost no possibility to write unsafe or malicious code as the functionality of the operating system is fully encapsulated within the virtual machine. Errors can be efficiently handled through exception management and the integrated logging subsystem provides an excellent facility for debugging. Furthermore, many advanced development and testing tools for Java, such as the Eclipse platform[2], are freely available.

7.3.1 Using the FermaT Transformation Engine

The next step in the implementation was to gather information of how the FermaT transformation system can be controlled. FermaT controls and applies its transformations through a special tool called the transformation engine or FermaT engine. The FermaT engine itself is implemented in Scheme and runs as a command line based tool. After downloading, extracting from the source package and compiling the user can command the FermaT engine via an interactive command line shell by running the file `MinGW\scmfmt.exe`. The user can now enter commands to control the

[2] http://www.eclipse.org

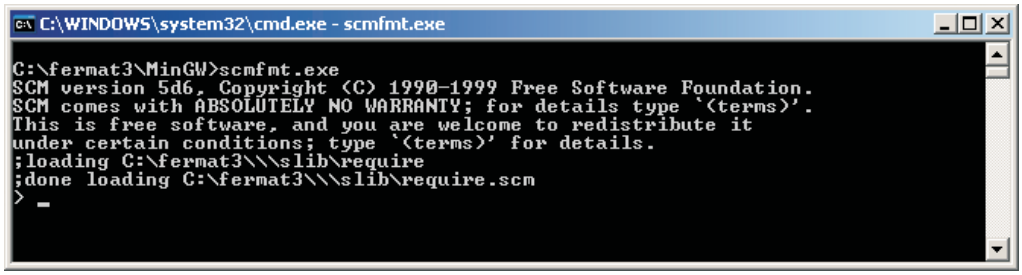

Figure 7.3: FermaT Command Line

transformation process of a WSL program. The first step is to load a WSL file into the engine. The following command loads the file `example.wsl`:

```
(@New_Program (@Parse_File "test.wsl" //T_/Statements))
```

Scheme is a LISP derivative and as such requires the writing in brackets. Scheme is case-insensitive by default which requires that all capital letters in names are preceded by a "/". Furthermore all WSL symbols are preceded by "/" (to avoid clashes with Scheme symbols) which makes it compulsory to write a "//" before all WSL names which start with a capital letter. After the command above has been send, the FermaT engine parses the WSL file and creates the Abstract Syntax Tree of the program in the memory. The tree can be printed to the screen with the command:

```
(@Print_WSL (@Program) "")
```

Every node in the tree has a general type (e.g. T_Condition for nodes used in condition statements) which identifies the node group and a specific type which identifies the node itself (e.g. T_True for the condition statement `TRUE`). In order to apply a transformation, the FermaT engine has to know on which specific node in the Abstract Syntax Tree the transformation should be executed. This is realised through a pointer which points to a single node in the Abstract Systax Tree called the *current item*. The position of the current item can be set with the `@Goto` function and retrieved with the `@Posn` function. By using the *current item* a transformation can be applied via the `@Trans` function. The following command applies the transformation *Reverse If* on the current item:

```
(@Trans //T/R_/Reverse_/If)
```

The function has also an optional string parameter which can contain additional data for a transformation. If the transformation is successful the FermaT engine would acknowledge this by returning `#t`. In case of failure the engine would print an error message and return with `#f`. In order to avoid the failure of a transformation and sometimes even unpredictable results, it is advisable to test the applicability of a transformation before use with the `@Trans?` function. The modified program can be written back to the file system after the transformation process has finished using the WSL Pretty Printer with the `@PP_Item` function. The Pretty Printer parses the FermaT engine's Abstract Syntax Tree and transforms it into source code. The following command would create the file `example_result.wsl` from the transformed Abstract Syntax Tree:

```
(@PP_Item (@Program) 80 "example_result.wsl")
```

With the knowledge of these commands it is possible to write the FME as a controlling graphical application for the FermaT engine, navigating through a loaded WSL program and applying transformations on individual nodes.

7.3.2 Connecting the FME to the FermaT Engine

Since the FermaT engine is a command line based tool it is possible to establish the communication between the FME and the engine via a communication pipe which is provided by the operating system. A software pipe chains a process to another process by one or more of its standard I/O streams. While a normal communication to a process is usually done through the keyboard and the

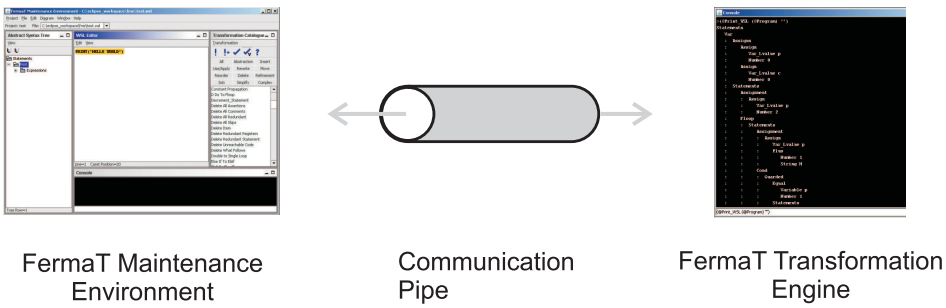

Figure 7.4: Communication between the FME and the FermaT engine

screen it is possible to redirect these streams of data to another process through a communication pipe. In this case the application is controlled by another process and not by the user. The output

of every program is usually divided into two separate streams called *input* for normal output and *error* for error messages while the input to the program is consequently provided through the *output* stream[3]. The FME starts the FermaT engine through the start script (`fermat_console`) which sets necessary environment variables before executing the FermaT interpreter (scmfmt). To create the actual process the FME uses the *exec* method of Java's Runtime class:

```
consoleProc = Runtime.getRuntime()
  .exec("engine/fermat_console.bat␣noecho");
```

Listing 7.1: Start of FermaT Engine (Windows)

```
consoleProc = Runtime.getRuntime()
  .exec("engine/fermat_console.sh␣noecho");
```

Listing 7.2: Start of FermaT Engine (Linux)

The *exec* method returns an instance of the Process class which can be used to access the streams of the software pipe:

```
in = consoleProc.getInputStream();
err = consoleProc.getErrorStream();
out = consoleProc.getOutputStream();
```

Listing 7.3: Communication Streams of the Pipe

The use of software streams is a useful technique to control console based programs via another process. The advantage is that the controlling application does not need to interact with internal processes of the controlled application. However, the disadvantage is that the communication over a pipe is slow compared to direct memory access. For an user interactive application this is not a big problem as human controlled input is usually not time critical (e.g. after the user has applied a transformation he needs time to evaluate the result before he proceeds). The communication latency only becomes a problem if many interactions are needed in a short time (e.g. if a fully automated application applies a transformation and needs only a few milliseconds to evaluate the result before applying the next transformation). In these cases it might be a good solution to either apply several transformations in a row without evaluating the result in every step or, if evaluation in every step is needed, to split the program into code chunks and apply the transformations

[3]The input stream refers, in this context, to the input which comes from the program to the user or other process while the output stream, on the other hand, refers to the data which comes from the user or other process to the program.

in parallel. This, however, needs a carefully designed algorithm which can decide how a given code and transformation sequence can be splitted so some transformations of the transformation sequence can be run in parallel.

7.3.3 Extracting the Transformation Catalogue

Before the FME can effectively use the FermaT engine it is necessary to extract information about the available code transformations. The bank of transformations is one of the FermaT engine's core components. As the engine itself also the transformations are written in $\mathcal{META}$WSL. The source code of the transformations comes with the software package and can be found in the `src/trans/` directory. Two files exist for every transformation: the source file (e.g. `elsif_to_else_if.wsl`) and a description file (e.g. `elsif_to_else_if_d.wsl`). The description files in particular is very useful for extracting information about the available transformations as it contains the name, some keywords and a description of what the transformation is doing. The keywords can be used to put the transformations into groups like "Rewrite", "Move", "Abstraction", etc. For the FME the information of all transformation description files where gathered and combined into a full list (see appendix C.1 and C.2). This list is used in the FME to generate a catalogue of transformations which the user can use to either test or apply a transformation or to get its description.

7.3.4 Extracting and Generating Data Structures

As the graphical user interface should show both, the Abstract Syntax Tree and the WSL source code, the next problem was to connect these two program presentations so the user is able to select nodes in the Abstract Syntax Tree by clicking on a code statement. Having only the Abstract Syntax Tree from the engine it is not possible for the FME without further analysis to determine which WSL code statement from the source file is represented by a particular tree node from the Abstract Syntax Tree. The solution is to generate the source code directly from the Abstract Syntax Tree while the actual WSL file is only read by the FermaT engine. This method has the advantage that transformation results can be shown in source code without writing the result to a file. The code generation in the FME is currently done in 3 steps:

1. Read the current representation of the Abstract Syntax Tree from the FermaT engine with the "`(@Print_WSL (@Program) "")`" command.
2. Parse the tree and generate a tree structure of objects which implement Javas TreeNode interface.
3. Print the tree by calling a special print method in every tree node object.

The code is thereby not constructed as a sequence of characters but as a sequence of lexical tokens. A lexical token is an abstract object which represents a whole keyword, separator, constant or identifier in the code (see section 3.4). The advantage is that keywords are always written correctly and that the task of selecting code segments becomes easier. Another usage for lexical tokens is the analysis of newly entered source code. If the user for example modifies the program in the WSL editor and wants to save his changes, the FME can check the syntax for validity by generating a sequence of lexer tokens form the newly entered source code. If a lexer token can not be generated for a sequence of characters then the user has made a mistake. For the implementation of these functionalities it was necessary to build a full list of all possible tree nodes and lexer tokens. Unfortunately such lists did not exist in the first place and all information had to be directly extracted from the source code of FermaT.

Lexer Tokens of Source Code

A WSL source code contain up to 195 different lexer tokens. A lexer token can be:

- white space characters (e.g. space, tabulator or new line)
- special characters (e.g. `+`,`*`,`;`, etc.),
- reserved words (e.g. `TRUE`,`IF`,`FOR`, etc.) or
- special tokens (e.g. `->`,`==`,`\/`, identifiers etc.).

Tokens can be defined either as a character sequence (used for keywords) or as a sequence of numbers which represent the ASCII codes of characters (used for all other characters). An ID number is only assigned to tokens which have a semantic meaning. Tokens like white spaces or

delimiters which do not have an ID are usually discarded after the lexical analysis is complete. A full list of all possible lexer tokens can be found in appendix B.4. The list is stored in XML format and contains the following information:

Fieldname	Description	Example
LexicalToken	The name of the lexical token.	`S_AND`
GroupName	The type of the lexical token.	`ReservedWord`
Characters	The characters which identify the token.	`AND`
ASCII-Code	The ASCII codes which identify the token.	
ID	The ID number of the lexical token.	`31`

Table 7.1: Columns of the Lexer Token List

If source code has to be analysed (e.g. if a WSL file is modified in the editor of the FME) and a list of lexer tokens should be generated from source code the FME uses a special lexer table which defines certain checking rules for the token identification. The FME lexer analyses the source code character by character assuming that tokens are normally separated by white space characters (space, tab or new line). Each rule checks for a certain character and if such character is found it identifies either the token directly or continues with the evaluation of subrules. The character checks of the FME lexer can be seen as a tree. Consider for example rule number 18 for identifying the signs not equal (“<>”), lesser equal (“<=”) or lesser (“<”):

1. If the current read character is a < sign.
   - 1.1. If the next character is a > sign. → Identify token as `S_NEQ` (not equal).
   - 1.2. If the next character is a = sign. → Identify token as `S_LEQ` (lesser equal).
   - 1.3. otherwise identify token as `S_LANGLE` (lesser).

The full rule table (see appendix B.2) has 30 main rules and 72 in total (including subrules). The list contains the following information:

Fieldname	Description	Example
No	The main rule number	`18`
CLevel	The checking step (counts upwards) mainrules start with 0 subrules with 0.1 and subsubrules with 0.1.1 .	`0.2`
Ident	If set to 1 then the rule is always true (used for the "otherwise" clause).	
Char	Used to identify a character (as plain text).	`=`
ASCII-Code	Used to identify a character (as ascii code).	
PassingRegularExpression	Used to identify a character (as regular expression).	
PassingGroup	Used to identify a character (as a group from the lexer token list).	
Identified Value	Identified token when check result is positive.	`S_LEQ`

Table 7.2: Columns of the Lexer Table

With these two tables it was possible to write a highly adaptable WSL lexer, able to parse every valid WSL file. Through the use of tables which are exchangeable it is also possible to reuse the written code to check the WSL dialects which are used within the Wide Spectrum Type System.

Nodes of the Abstract Syntax Tree

WSL has 3 types of tree nodes: General type, Group type, Specific type. Each *specific type* represents a concrete code statement (e.g. a minus sign "-" is represented by the tree node T_Minus). These nodes are structured into classes according to the places in a program where they can be used. A *general type* is a tree node which represents such a classification of normal nodes. Every specific type which is in their class starts with their ID number e.g. T_Minus (No. 221) belongs to T_Expression (No. 2) because it is only used within expressions. This classification is necessary to constrain the possible statements in an expression (e.g. the condition of an `IF` statement must be done with nodes which belong to T_Condition). There are 9 general types: T_Statement,

T_Expression, T_Condition, T_Definition, T_Lvalue, T_Assign, T_Guarded, T_Action, T_Name. To be able to write sequences of certain statements there are also 7 group types (e.g. most WSL programs are a sequence of statements in which case the first node in the Abstract Syntax Tree is T_Statements): T_Expressions, T_Lvalues, T_Assigns, T_Definitions, T_Actions, T_Guardeds, T_Statements. The final list of all tree nodes can be found in appendix B.3 and B.4 (continuation). The list is used by the FME to parse, validate and process a generated Abstract Syntax Tree from the FermaT engine. The information from the list is stored in the tree nodes of the FMEs internal tree structure. Every tree node of the tree structure contains the following information:

Fieldname	Description	Example
ID	The ID of the node type.	`221`
Name	The name of the node.	`T_Minus`
Syntax Name	The Syntax Name of the node.	`Minus`
General Type	The general type of the node.	`2`
Allowed children	The allowed child nodes.	`2`
Has Value	If the node can have a value.	`0`
PrettyPrintTemplate	A template for the pretty printer	`S_LPAREN;(A, #S_MINUS# );S_RPAREN`

Table 7.3: Columns of the Tree Node List

If a general node is mentioned in the list "Allowed children" than all nodes which belong to its class can be a child node. A node can either have children or a value i.e. if "Has value" contains a 1 then no child nodes are allowed. The information of the column PrettyPrintTemplate was added manually and states special print instructions for translating the Abstract Syntax Tree into source code. The instructions are separated by semicolons and can be one of the following:

Command	Description
SPACE	Print a single space character.
`I`	Increase the indention (added spaces at the beginning of each new line).
TOKEN	Print a lexical token (e.g. `S_Minus` would print a minus sign "-").
`CS:`TOKEN	Print a lexical token only if there is a sequence of child nodes i.e. if the node has more than one child node.
`(V)`	Print the value of a node.

C:NUMBER	Print an ASCII character (e.g. C:67 would print a "C".
(A,TOKEN#TOKEN)	Call the print method of all children and separate them with specific lexer tokens separated by a hash "#" (e.g. (A,S_COMMA#) would separate all children with a comma and a following space).
\n	Print a new line character.
(C0)	Call the print method of the first child node ((C1) for the second child node, and so on).
PARENT(NODE:TOKEN)	Print the lexical token under the condition that the parent node is of a certain type (e.g. PARENT(T_Cond:S_THEN) would print a "THEN" if the parent node is of type T_Cond).

Table 7.4: Pretty Print Commands

Example

The following example demonstrates the whole process of generating source code from the Abstract Syntax Tree with all involved processes. For this example the assignment I = (1 + b) was loaded into the FermaT engine and the following Abstract Syntax Tree was extracted using the @Print_WSL command:

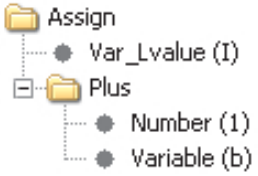

Figure 7.5: Example Abstract Syntax Tree

The tree is printed by calling the print method of the top node T_Assign. This method uses the PrettyPrint template of the Tree Node List to generate the source code from the Abstract Syntax Tree. The following rows of the PrettyPrintTemplate column are involved (taken from appendix B.4):

Name	PrettyPrintTemplate
T_Assign	(C0); ;S_BECOMES; ;(C1)
T_Var_Lvalue	(V)

T_Plus	S_LPAREN;(A, #S_PLUS#);S_RPAREN
T_Number	(V)
T_Variable	(V)

Table 7.5: Excerpt from the Tree Node List

For following table shows the order in which the nodes in the Abstract Syntax Tree are traveled and how the PrettyPrinter generates the source code:

Involved Node	PP Command	Description	Generated Code
T_Assign	(C0)	Call the print method of the first child node.	
T_Var_Lvalue	(V)	Print the value.	I
T_Assign		Print a space.	I
T_Assign	S_BECOMES	Print a lexer token.	I :=
T_Assign		Print a space.	I :=
T_Assign	(C1)	Call the print method of the second child node.	I :=
T_Plus	S_LPAREN	Print a lexer token.	I := (
T_Plus	(A, #S_PLUS#)	Print all children separated by " + ".	I := (
T_Number	(V)	Print a value.	I := (1
T_Plus	(A, #S_PLUS#)	Print all children separated by " + ".	I := (1 +
T_Variable	(V)	Print a value.	I := (1 + b
T_Plus	S_RPAREN	Print a lexer token.	I := (1 + b)

Table 7.6: Example Processing of the Pretty Printer

To connect now the tree with the code the FME simply stores the lexer tokens which were generated by a particular tree node. If the user selects a token in the code, then the AST node is selected which has generated this token (e.g. a click on the equals sign would select the T_Assign node). If the user clicks on a tree node, then all token are selected which were generated by this node and its children (e.g. a click on T_Assign would select the whole assignment).

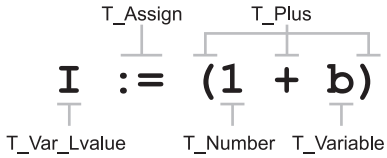

Figure 7.6: Example Code with Connected Tree Nodes

7.3.5 Interface of the FermaT Maintenance Environment

With a connection to the FermaT engine via the pipe, the information from the transformation catalogue and the possibility to extract data structures from a loaded program it is now possible to build the graphical user interface. The purpose of this environment is to ease the manual

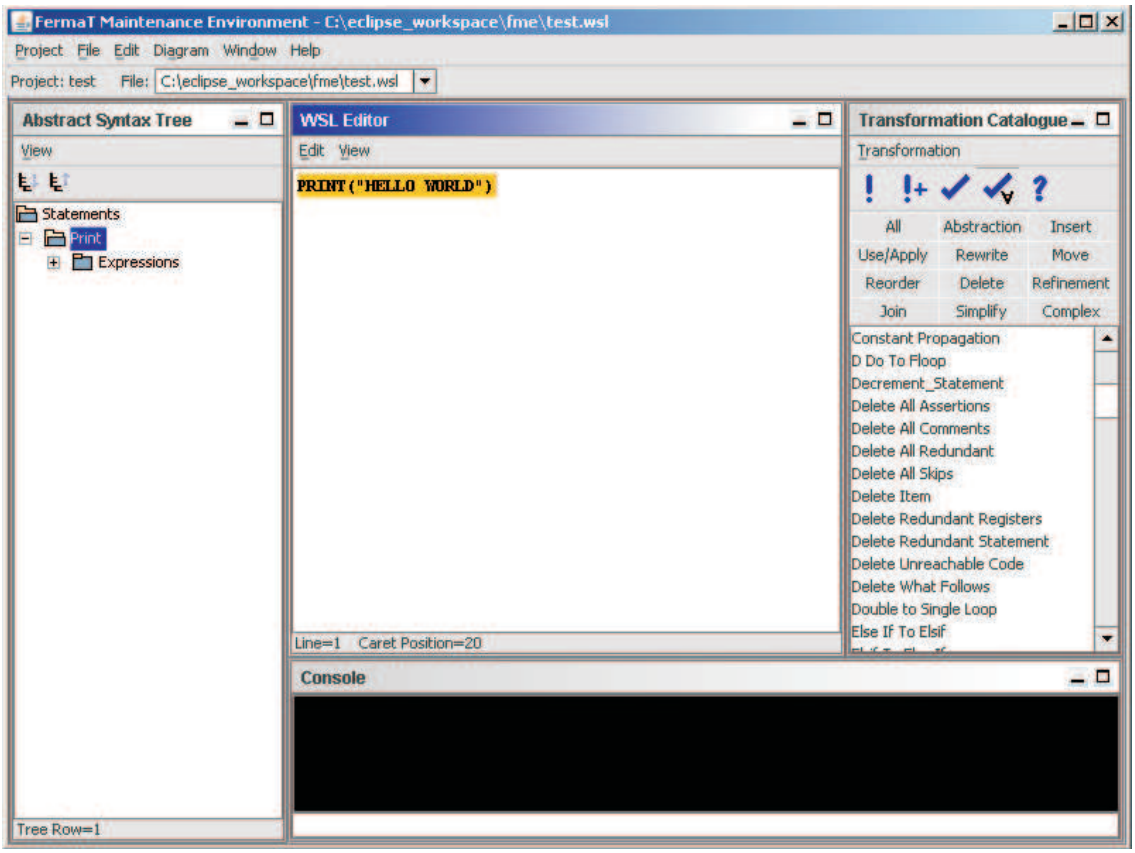

Figure 7.7: FermaT Maintenance Environment

transformation process and to support the development of other advanced tools based on the FermaT transformation system. Figure 7.7 shows the main window of the FME. It consists of a text editor which is able to express WSL and a viewer for the Abstract Syntax Tree which shows the code in the text editor as Abstract Syntax Tree. As described in section 7.3.4 the user

can click on a statement in the code to highlight a node in the Abstract Syntax Tree and vice versa. The environment provides furthermore a command console which can be used to directly enter commands to the FermaT engine through the communication pipe. A transformation can be chosen from the transformation catalogue and either tested or applied on the current selected node in the Abstract Syntax Tree. Should a transformation succeed but not result in an intended outcome, it is always possible to revert this action by using the the UNDO command. It is furthermore possible to group multiple files in a project if the migration involves more than one WSL file.

7.3.6 Visual Representation

To ease the task of program comprehension the FME supports the generation of diagrams. The diagrams are displayed with the STRL Visualisation Engine (SVE) which is a java library to display graph diagrams using the Zoomable Visual Transformation Machine (ZVTM)[4]. The SVE is build from extracted code of a former project [Lad06]. In its current implementation the FME supports Call Graphs for procedures and action systems (FermaT uses action systems to model GOTO structures) and a transformation history graph which shows the history of applied transformations within a project.

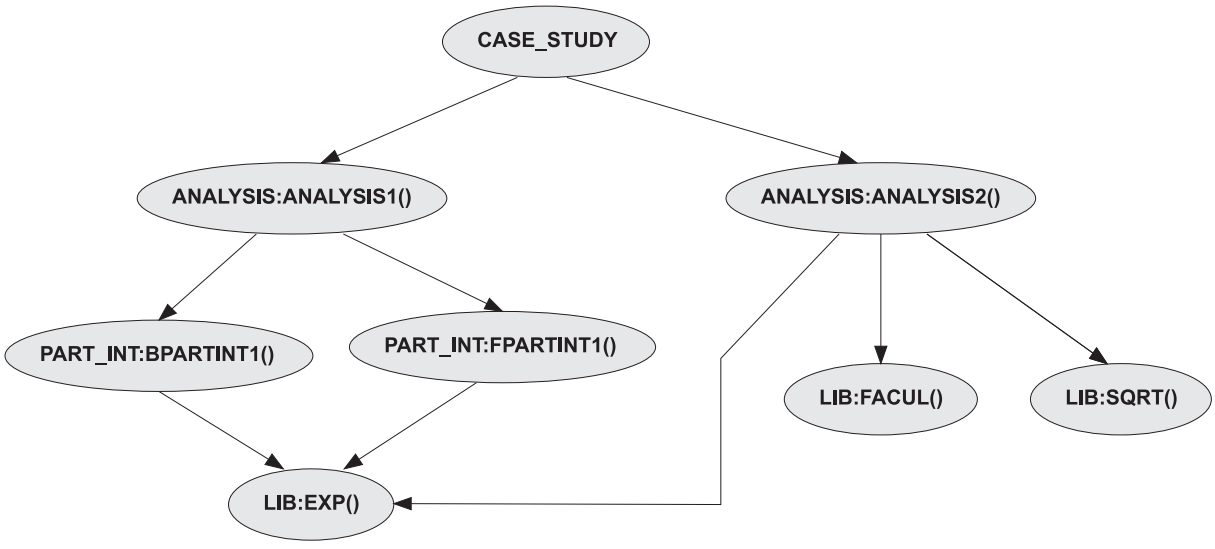

Figure 7.8: Example Call Graph (Procedures)

[4]See http://zvtm.sourceforge.net/ .

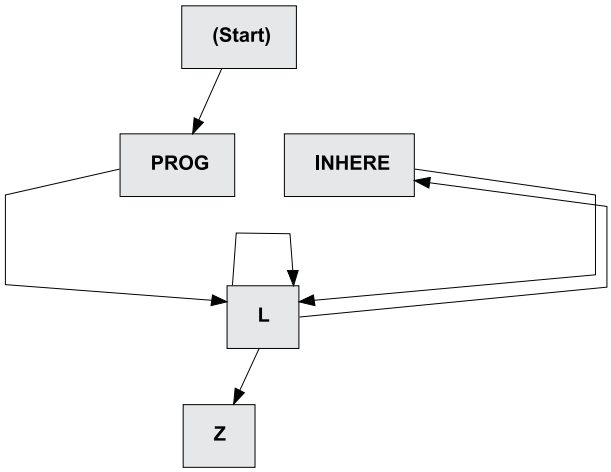

Figure 7.9: Example Call Graph (Action System)

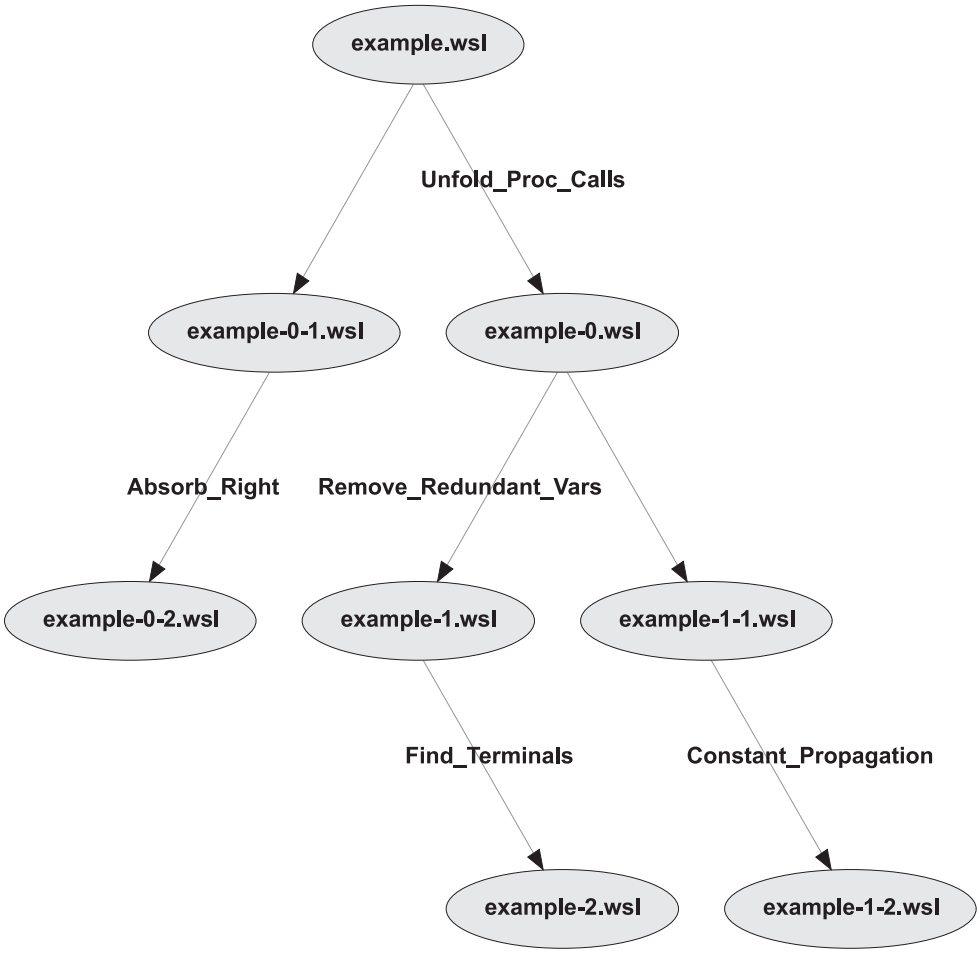

Figure 7.10: Example Transformation History Graph

7.3.7 Implementation

The implementation design of the FME is strictly modular. While the core part consists of around 10 classes, the whole implementation has more than 130 classes with a total of 4439 lines of code in 417 methods excluding the parsers for typed WSL and the SVE which are held in packed jar archives. The core components of the FME include:

- **Framework classes:**

  ComponentRegistry Manages all other sub-components. This includes object reference management, global event management for events which concern more than one object (e.g. loading a file, initialisation, etc.).

  EventTracker Records all modification events to a WSL file for the undo/redo function.

  IOManager Whenever a class wants to interact with the operating system, like reading or writing to a file, it uses the IO Manager for the actual operation.

  MainEventHandler Handles the events when the user interacts with the core components. It initiates functions like opening a WSL file/project, undo/redo, execute a WSL file, compiling and presenting diagrams.

  ProjectManager Handles the project management. A FME project is a group of WSL files for which a modification / transformation history is kept.

- **Logging classes:**

  FMELogger The logging system of the FME. All logging messages are handled here using the Java Logging API.

- **Configuration:**

  CM The configuration manager is the source for all configuration parameters. These parameters are stored in a XML configuration file and kept in a hash table during runtime. All component request their user specifiable parameters from this class.

- **Graphical User Interface:**

  MainFrame The main window of the FME with a menu and tool bar.

- **Datastructures:**

  AST Can parse the Abstract Syntax Tree from the the FermaT engine and keeps the generated data structure available for all other components. It uses the Tree Node List mentioned in section 7.3.4 to check and verify the correctness of its data structure.

  WSLLexer Performs the lexical analysis on any WSL file and generates a list of lexer tokens. It uses the Lexer Token List and the Lexer Table described in section 7.3.4.

All other classes which are not core components belong to so called sub-components. Sub-components are reusable discreet entities which control single, mostly visual, sub-components of the FME. All sub-components are sub-classes of `GUIComponent` which defines the management for internal frames and several standard access methods for the managing framework classes. The sub-components include:

- **GUI sub-components:**

  Transformation catalogue The transformation catalogue from which the user can select a transformation and either apply or test it. It is also possible to test for all possible transformations on a particular tree node and to get a description of what a particular transformation does.

  Graphical Abstract Syntax Tree The graphical representation of the Abstract Syntax Tree. This component visualises of what is held in the AST data structure component. The user can navigate through the tree by expanding or collapsing single branches of the tree.

  Console This component models the connection to the FermaT engine. Its graphical part shows a console which allows the user to type in command to the engine. However, the real communication over the pipe is exclusively done in its non-graphical part. This part handles also the communication with the engine of all other FME components.

  Editor The editor handles and displays WSL code. The user is able to edit the displayed code and to highlight statements or groups of statements by clicking on them. The editor includes also a search dialog in which the user can search or replace code via keywords or regular expressions.

- **Visualisation sub-components:**

  Function call graph This graph visualises a call graph which is a directed graph that represents calling relationships between functions. Each node represents a function and each edge indicates that a function calls another function. A cycle in this graph indicates a recursive procedure call. The user is able to click on a node to see the corresponding function within the WSL code and on an edge to highlight the call.

  Action system call graph Like the call graph this graph visualises calling relationships. But instead of functions the entities which are represented by the nodes are actions in an action system. An action system is a collection of mutually recursive parameterless, usually very small, procedures. The system starts by calling the start action, executing its body and calling via the "call" command the next action. This is repeated until the special action call "CALL Z" which will terminate the whole action system. Action systems are usually used to model GOTO structures. The user is able to click on a node to highlight the corresponding action.

  Transformation history graph This graph is only available within a project. It shows the transformations which have been applied so far to all containing files whereby each node is representing a version of a particular file and each edge represents a transformation or a saving to a different file. The user is able to click on a node to see its code and on a transition to highlight the node on which the transformation was applied.

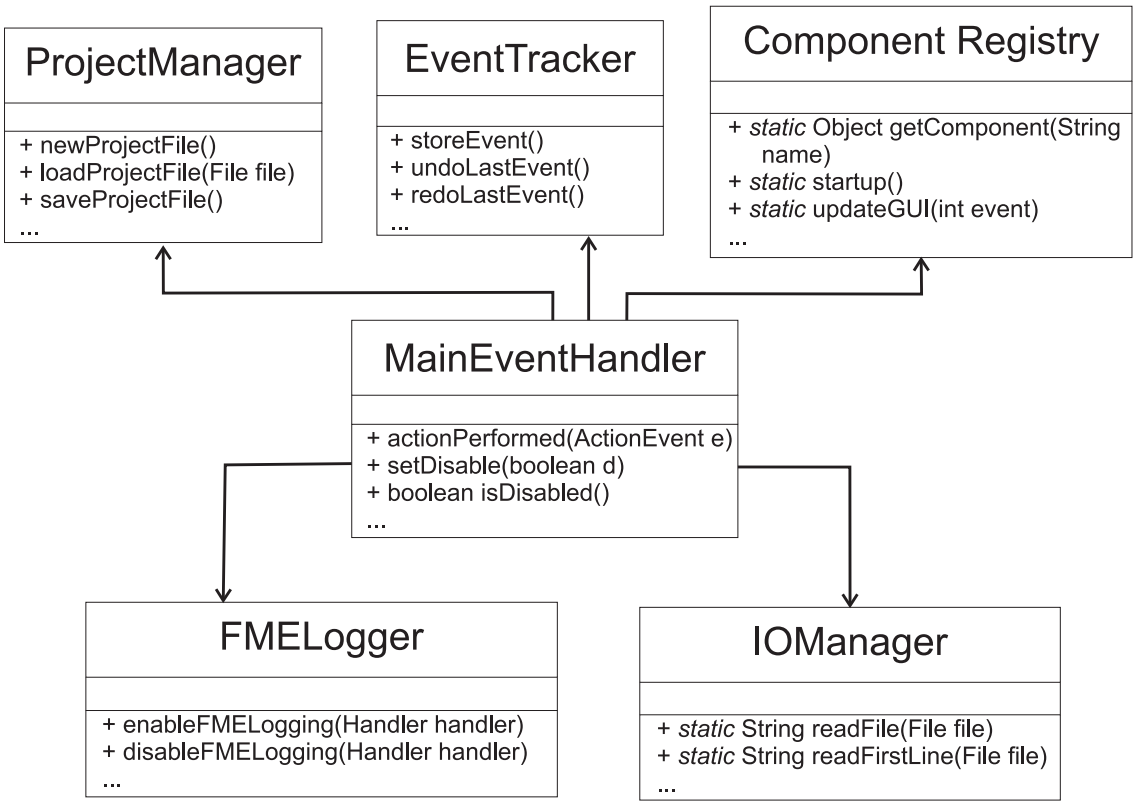

Figure 7.11: UML Diagram of Framework and Logging Classes

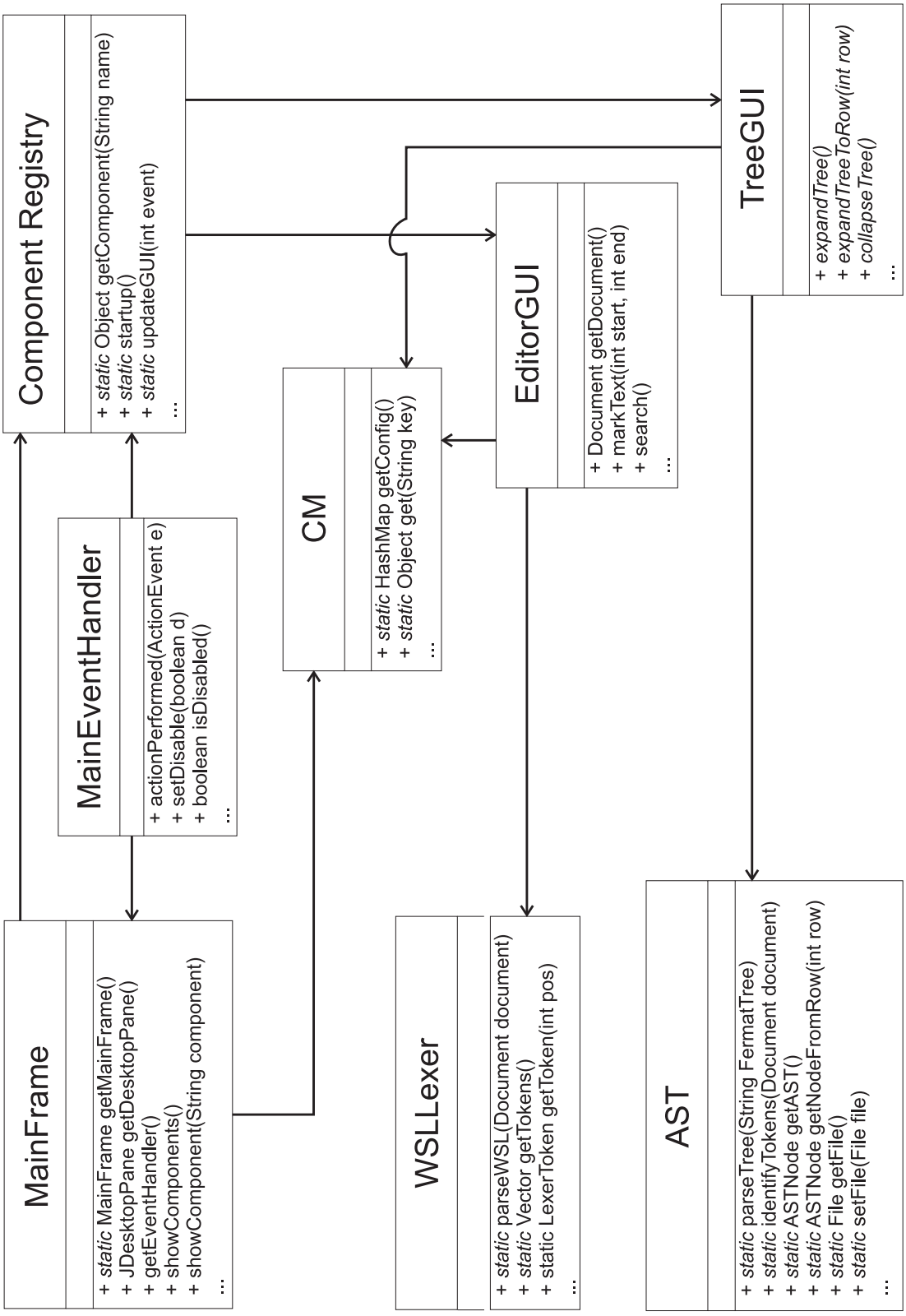

Figure 7.12: UML Diagram of configuration, GUI and Data Structure Classes

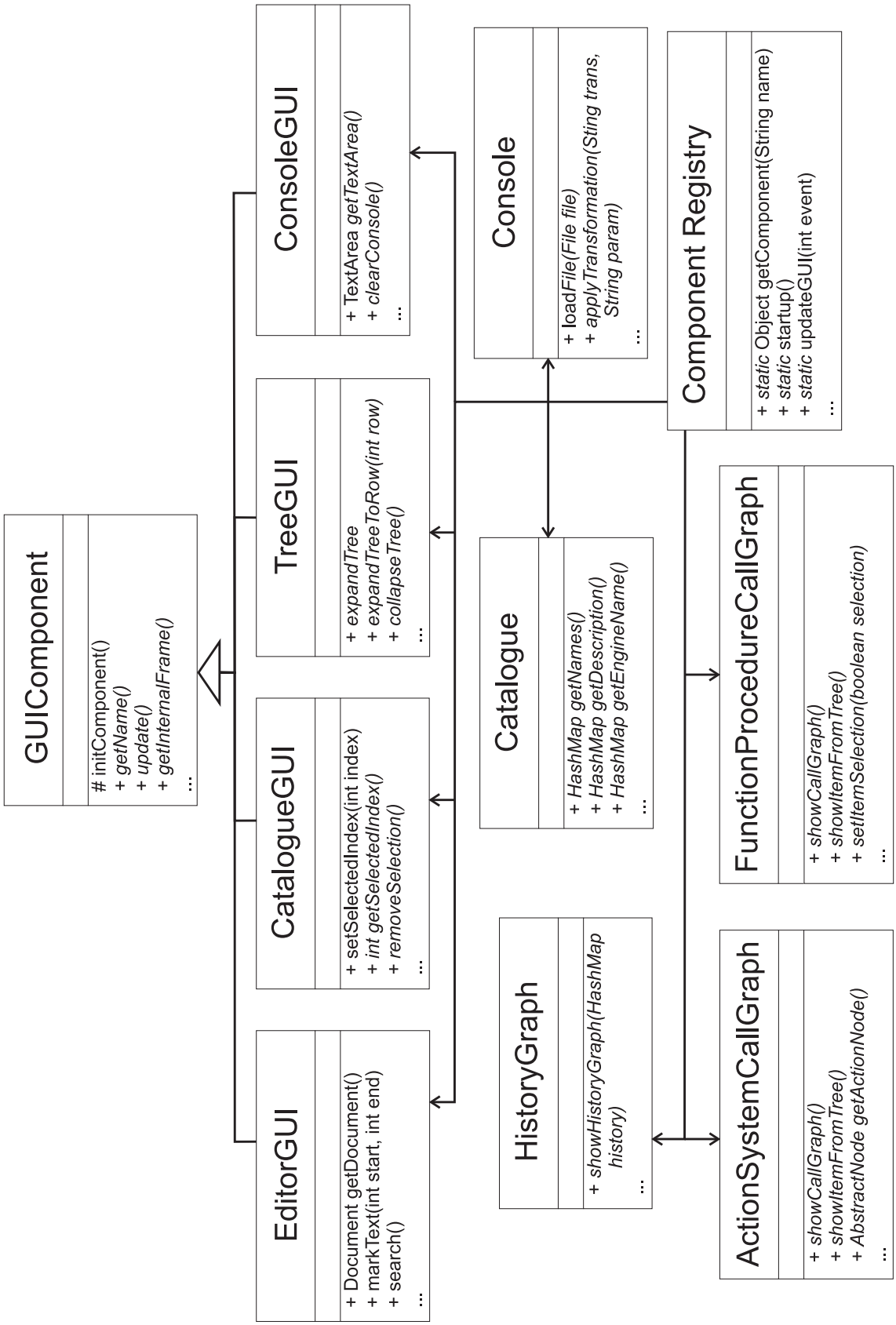

Figure 7.13: UML Diagram of Sub-component Classes

7.4 FermaT Type System Editor

The FermaT Type System Editor (TSE) is the first program based on the FermaT Maintenance Environment and the prototype implementation for the described research. Many ideas, design concepts and experience from the FME were incorporated and integrated into the TSE. The maintainer can use the Type System Editor for one of the following:

- to introduce a type system to an untyped program via type inference.
- to convert a type system of a program which had already a type system.
- to verify the type correctness of a program on a certain layer of the Wide Spectrum Type System.

The following use cases were identified for the TSE:

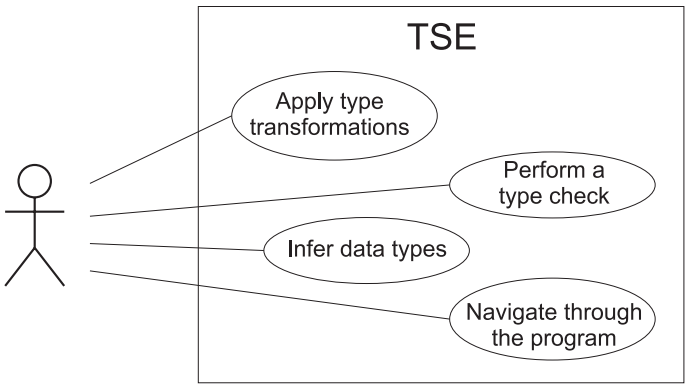

Figure 7.14: TSE UML Use Case Diagram

A major difference to the FME is that the WSL code is imported into the environment from a file rather than loaded. This means that the TSE only reads the code from a file and takes full control over it until it is exported into a file again. This makes sure that the code is only altered through the TSE and its type transformations which is important because many type transformations have to assume that a program is correctly typed as a type check after each type transformation would be too time consuming. However, a user can only do modifications, if the code needs to be changed due to found errors or inconsistencies, by using a special internal editor which checks the code for

correctness before allowing any further type transformations to be carried out. Another difference to the FME is the diversity of code navigation facilities. Especially in case of large-scale software systems, it is essential to provide several different possibilities for code navigation, to handle their complexity.

7.4.1 Variable List

The most important entities for a type system within a program are the variables. Therefore, the TSE features, besides the code navigation functionality of the FME via Abstract Syntax Tree and Source code, a special view which shows all variables of a program together with their type and their scope of definition. Such an overview can greatly assist in the task of program comprehension or when investigating typing errors. Figure 7.15 shows an example of a program and its variable list. The information needed for such a list can be gathered by parsing the program and recognising

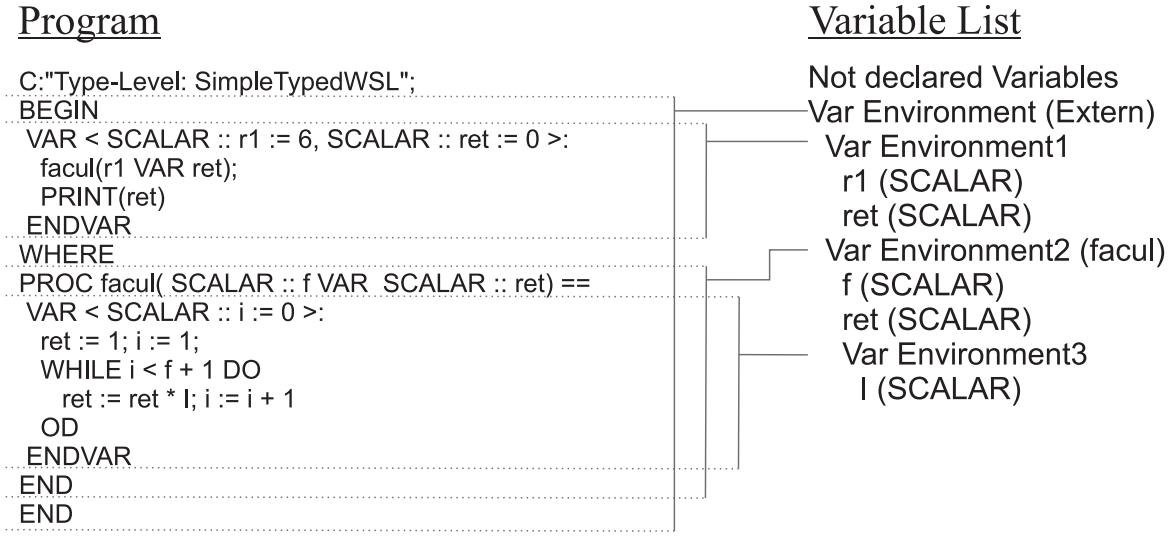

Figure 7.15: Program and Variable List

all used variables and the opening and closing of scopes. In WSL a variable scope is mainly defined through a VAR environment or through a sub-routine. In the variable list these scopes are presented as numbered *Var Environment*s. The indention shows whether an environment is enclosed in another environment (e.g. Var Environment 3 in Var Environment 2) or if an environment is completely distinct from another environment (e.g. Var Environment 1 and Var Environment 2).

7.4.2 Extracting and Generating Data Structures

Hence the Type System Editor supports all typed versions of WSL it is completely separate from the FermaT engine. It was necessary to rewrite all of the FermaT engine's code analysing routines for creating the Abstract Syntax Tree in Java. For the untyped and all typed versions of WSL it was required to create a separate lexer token list, lexer table and tree node list and also a code parser which does the actual generation of the Abstract Syntax Tree. Especially the last part is very critical and requires normally extensive coding[5]. However, since the parsing of source code is one of the best researched areas in computer science, there are many supporting tools available [PP92]. Using java, there is no need to write the actual parser by hand as there exists a powerful parser generator called JavaCC[6] which is able to create a parser by reading a definition file, written in a special form of BNF. Using this tool, it was sufficient to write such a BNF definition for all typed WSL versions. With this parser integrated into the Type System Editor, it was possible to reuse many data structure related algorithms which were originally developed for the FME. An example JavaCC definition for the Untyped WSL version can be found in appendix B.5.

7.4.3 Interface of the Type System Editor

The interface of the TSE features 3 main windows. As the FME also the TSE has a viewer for the Abstract Syntax and a code editor for the source code itself. The transformation catalogue and the console were replaced by a variable list and the views of all three windows were connected. This means that a click on a variable in the code also highlights the variable in the Abstract Syntax Tree and in the variable list while a click on a variable in the variable list highlights all occurrences of this variable in the source code. Searching is available for the Abstract Syntax Tree, the source code and the variable list. Type transformations, type inference and type checking are available from a popup catalogue.

[5] The parser within the FermaT engine has more than 2800 lines of code.
[6] https://javacc.dev.java.net/

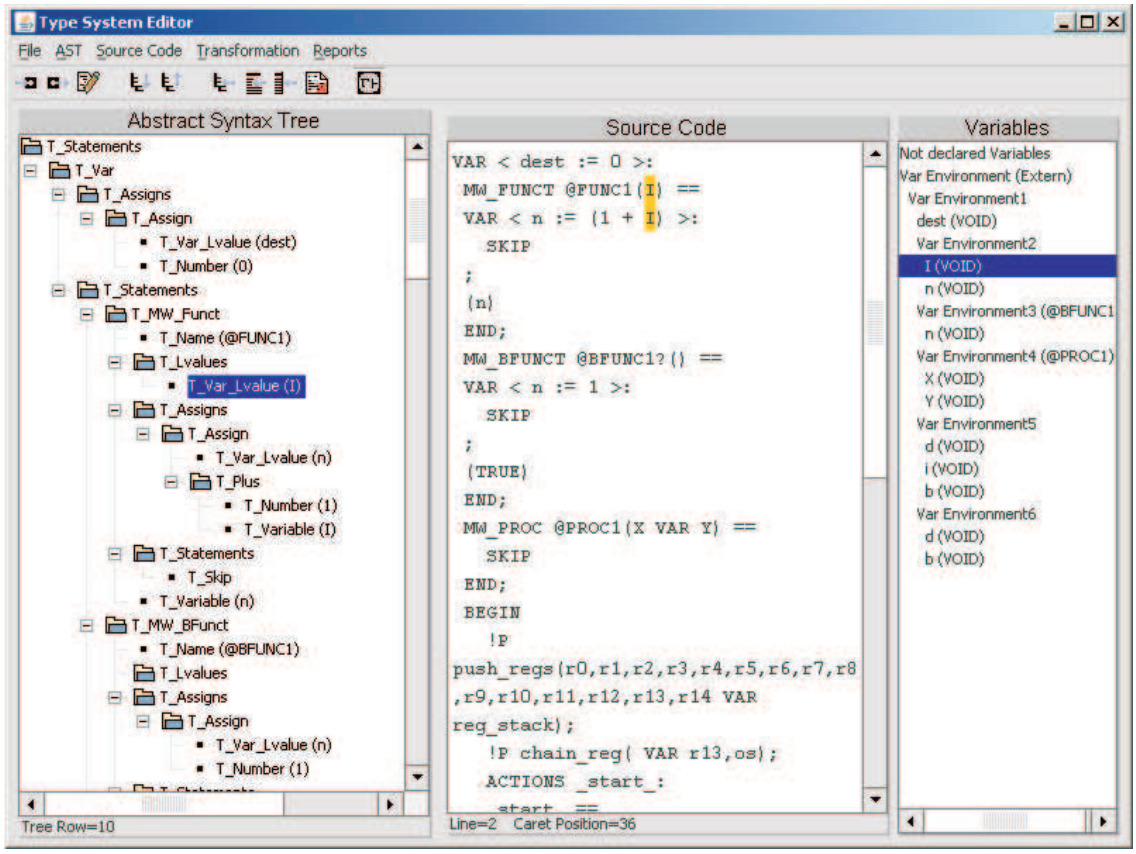

Figure 7.16: Type System Editor

7.4.4 Implementation

Although the TSE is not as modular as the FME, the design still emphasises the reusability of important main components. Especially the type checking / inference algorithm and the type transformations can be easily reused. Many Type System Transformations, which are described in section 6.5, are already implemented together with the adjustable type checking and type inferencing algorithms. The core components of the TSE are:

- **Controlling classes:**

  TypeSystemEditor The main class which initialises and starts all other components.

  TypeSystemEditorListener Controls the whole interaction process with the user.

- **GUI classes:**

TSEMainFrame The main window which controls all other GUI components. It also displays the graphical representation of the AST Tree.

TSEVarList The variable list.

TSEEditor The editor.

TSECatalogue Controls the transformations and type checking / inference algorithms.

- **Algorithms and Data Structures:**

TSEInferencer Implements the type checking / inferencing algorithm described in section 6.2.

TSELexer Reads any valid WSL file (typed or untyped) and creates a sequence of lexer tokens.

TSEParser An interface to the JavaCC generated parsers for untyped and typed WSL.

TSEAST Holds the Abstract Syntax Tree of the current processed program.

TSELTL Holds the lexer tokens of the current processed program.

TSEVarList Collects and holds all information displayed within the variable list.

Figure 7.18 shows the integration of transformations within the TSE. Although the process of type checking / inference is distinct from a type transformation, both algorithms can be seen as a transformation on the implementation level. Although the type check or the type inference algorithm do not change the source code, they do change the data structures i.e. they create additional information within the Abstract Syntax Tree. In the implementation of the TSE the process of type inference and type checking are therefore regarded as transformations which do not change the source code but which change the Abstract Syntax Tree and report an either positive or negative result. In contrast to the FermaT transformations which are executed on single nodes in the Abstract Syntax Tree, the TSE transformations are always executed on the whole program. Every time a transformation is successfully executed a new file is generated and loaded into the TSE. If a transformation fails or cannot be applied a negative result is returned and the transformation process is terminated. All transformations are sub-classes of `AbstractTransformation` which defines essential procedures like `checkTransformation()` or `applyTransformation`. In this implementation many complex transformation usually depend on one or more basic transformations which

builds a hierarchy of transformations. An example for this is the transformation InsertTypeAnnotations which uses information generated by InferTypes to insert variable declaration with a suitable type for every variable.

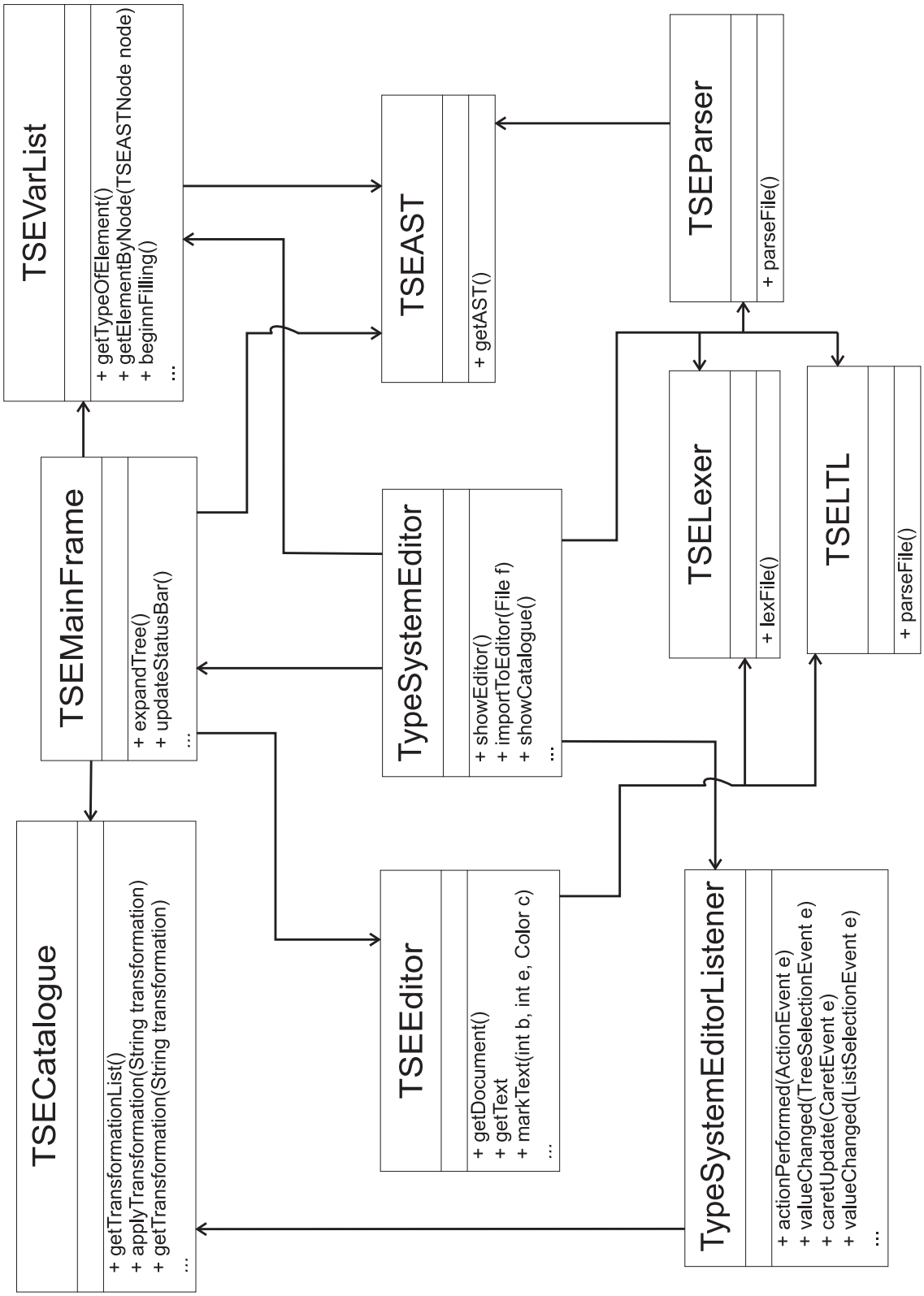

Figure 7.17: UML Diagram of Controlling, GUI, Algorithms and Data Structure Classes

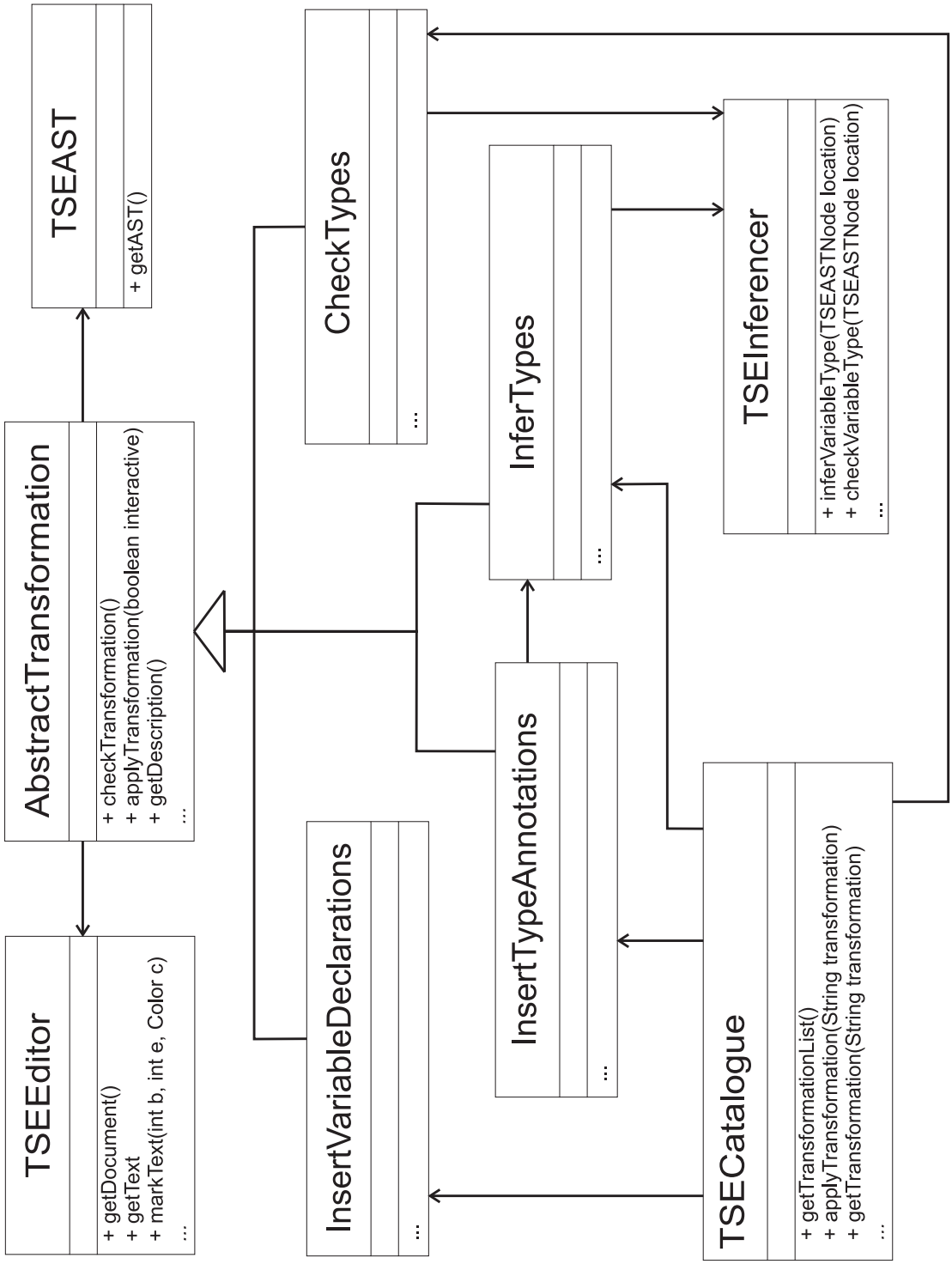

Figure 7.18: UML Diagram of Algorithm Classes (cont.)

7.4.5 Integration into the FermaT Transformation Process

The type checking facilities of the implementation are designed to integrate seamlessly with the current migration process. As described in section 2.10 the current implementation of the FermaT migration process has to store the information about data types in a separate file which causes several drawbacks. Only some parts of the current migration process can be proven because many process steps are based on at least a few assumptions. The FermaT migration process uses a collection of scripts which handle the migration steps. Listing 7.4 shows a standard log file of a IBM 360 assembler to C migration.

```
FermaT Migration Assembler to C.
RCS $Revision: 1.5 $ $Date: 2004/09/15 13:02:07 $
Started by root at Sat May 31 01:13:50 2008
Files to process in this run:
FMT001A0.lst

FMT001A0.lst (1/1)
-- a2w "FMT001A0.lst" "FMT001A0.ws1"
-- metrics "FMT001A0.ws1" "FMT001A0.me1"
-- datreorg "FMT001A0.dat" "FMT001A0.da2"
-- dat2ll "FMT001A0.ws1" "FMT001A0.da2" "FMT001A0.ll"
-- dat2c "FMT001A0.da2" "FMT001A0.hh"
-- split-headers "FMT001A0.lst" "FMT001A0.hh" "FMT001A0.h"
-- split-headers FMT001A0.lst "FMT001A0_init.hh" "FMT001A0_init.h"
-- dotrans  "FMT001A0.ws1" "FMT001A0.ws2" Find_Dead_Code
-- dotrans  "FMT001A0.ws2" "FMT001A0.ws3" Data_Translation_A
   data="FMT001A0.ll"
-- dotrans  "FMT001A0.ws3" "FMT001A0.ws4" Fix_Assembler data=4000
-- dotrans  "FMT001A0.ws4" "FMT001A0.ws5" Delete_Redundant_Regs
   data="FMT001A0.ll"
-- dotrans  "FMT001A0.ws5" "FMT001A0.ws6" Data_Translation_A
   data="FMT001A0.ll"
-- dotrans  "FMT001A0.ws6" "FMT001A0.ws7" Fix_Parameters
   data="FMT001A0.ll"
-- metrics "FMT001A0.ws7" "FMT001A0.me7"
-- wsl2c "FMT001A0.ws7" "FMT001A0.raw"
-- tidy_c "FMT001A0.raw" "FMT001A0.c"
-- /usr/bin/gcc -I . -I /fermat2/config -c "FMT001A0.c"
... processed FMT001A0 to Level 5 in 4 secs,
```

```
    Tue September 16 18:24:01

SUMMARY:

     1 files processed to Level 5

--------------------------------

     1 files processed altogether

--------------------------------
```

Listing 7.4: Migration Log

Figure 7.19 shows this process in a graphical form. The FermaT engine accepts assembler listing files and outputs the C source file with the code, the C header with the declaration of all data types and a special C source file for initialisation which initialises the data. The initialisation of data in

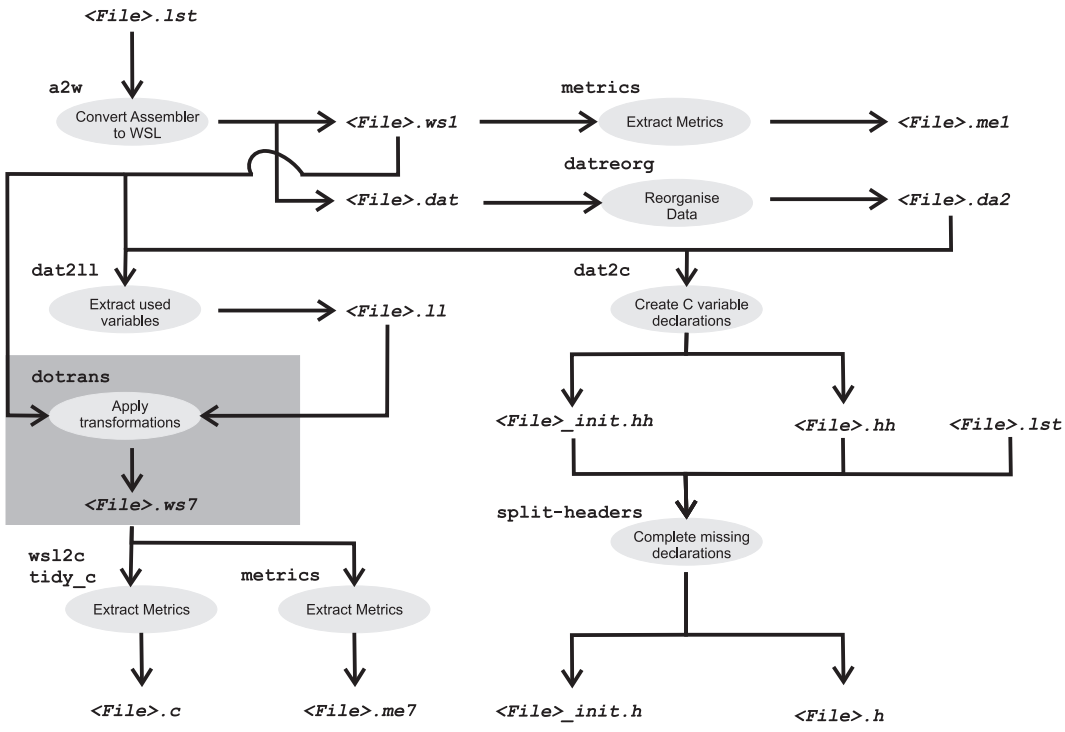

Figure 7.19: Current Migration Process to C

a separate source file is a necessary workaround as C cannot initialise the contents of a STRUCT data type upon declaration. The gray circles describe migration steps with the handling script in the upper left corner. Each script has one or more input file(s) and one or more output file(s). The

steps “Extract metrics” do not really contribute to the migration result as they only measure the complexity of the program with certain metrics. However, they are included in the graphic to fully illustrate the complete process. The dark gray box describes the part of the process which can be proven to be correct. The purpose of this graphic is to show how many steps within the migration process are based on assumptions and cannot be proven. Especially the translation of data is most dangerous as it is directly translated into the target language. The proposed integration of the type checker for the Wide Spectrum Type System into the migration process is presented in figure 7.20. Notably the provable parts of the migration process increased as the type checker can

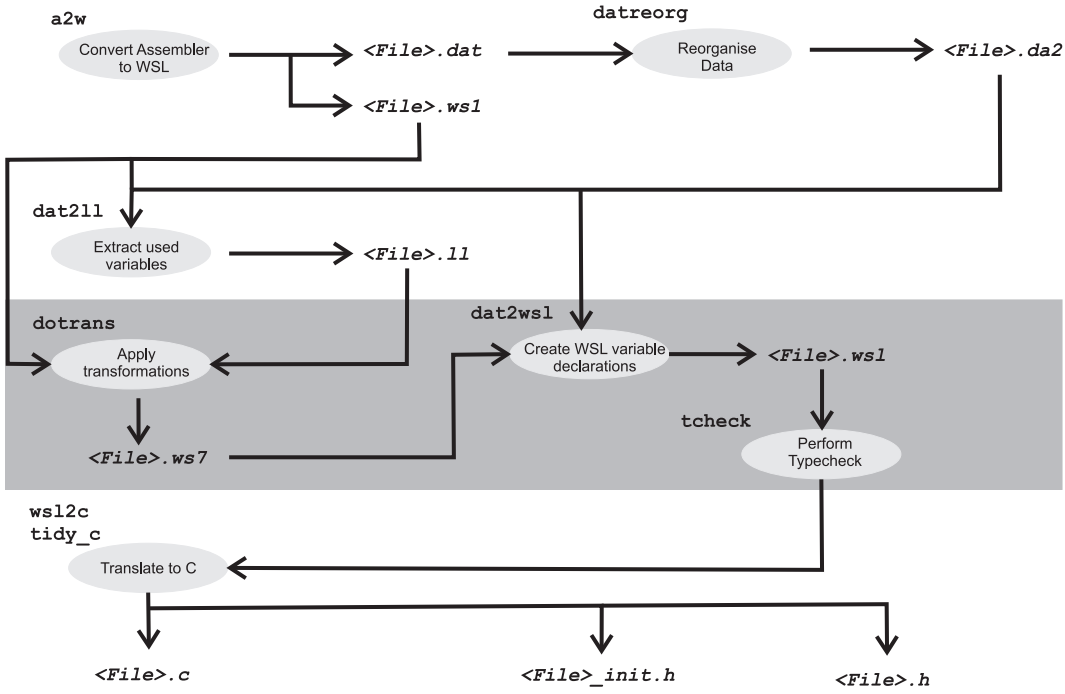

Figure 7.20: Enhanced Migration Process to C

verify that the translated data structures do indeed work with the code in the designed target language. It is not necessary to rewrite any transformation or to revalidate any of their proofs as the introduction of types and their verification are carried out after the code has been abstracted. In this way the approach only adds an additional verification mechanism to correct the flaws of the migration process. An important fact of this approach is that the new verification mechanism is done in WSL and not by a compiler in the target language. The reason for this is the fact that a translation process always introduces at least a few assumptions which can cause the system to malfunction. A good example for this is presented in the first case study in section 8.2 which involves a migration from C to FORTRAN. The problem occurs due to an implicit type cast in

combination with a translated call-by-value to call-by-reference procedure invocation.

7.5 Summary and Conclusion

This chapter reviewed and describes the tools which have been developed during this research investigation which are mainly the FermaT Maintenance Environment and the FermaT Type System Editor. These tools are still in the conceptual prototype stage. Although they have been tested among some case studies, the way before these tools can be considered for an industrial production environment is still long. Especially errors related to the underlying operating system are still quite common. Nevertheless, the fundamental conceptual parts of development have been done and the resulting prototype application is working and useful. So far the development includes the following:

- Lexer and parsers for all WSL versions.
- A consistency check for WSL Abstract Syntax Trees.
- Pretty Printer for all WSL versions.
- The grammatical definition of all WSL versions.
- Typing rules for all type system layers.
- The implementation of the type checking and type inferencing algorithm.
- An extensive java API to navigate through Abstract Syntax Trees.
- An initial set of type system transformations.
- An initial object identification algorithm.

The development was done in cooperation with the company who funds the research and the resulting prototype has been successfully tested on Linux and Windows operating systems by many students during a Master of Engineering course as practical part of a Software Evolution module.

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Ervin Knuth

Chapter 8

Case Studies

Objectives

- To give a small scale case study of an unsafe procedural system which is migrated into a safe procedural- and object oriented code.
- To give a medium scale case study of an procedural system which is translated into a object oriented code.
- To demonstrate and evaluate the need and practical applicability of the presented research.

8.1 Introduction

This chapter presents two case studies to illustrate the practical applicability of the presented research. The first smaller case study migrates code which was taken from a bigger collection of algorithm for numerical analysis and demonstrates why a migration process must pay special attention to the correct capturing of typing. The second case study is a migration of a medium scale software system which demonstrates the robustness and scalability of the approach. Both case studies includes besides the correct conversion of data types also the identification of objects and the transformation from procedural into object oriented code[1].

[1]The source code of all translation steps of case study 1 can be found in appendix A.2.

8.2 Case Study 1

The presented case study is a small excerpt of a bigger software system to demonstrate certain aspects of the proposed approach in more detail. The case study was chosen because it models quite complex mathematical functions with only a few lines of code which is written in C with a particularly interesting usage of data types. The software was utilised for practical demonstration of algorithms to students during a numerical analysis course to presents iteration based algorithms; two partial integration algorithms for $\int_0^x t^n \cdot e^{-t} dt$ and one square root function. The case study is fully written in C and consists of four modules. It has the two global variables: *debug* which is a switch to toggle the output of debug messages and *MAX_IT* which tweaks the speed and accuracy of the square root function. During this case study the system will be safely translated to FORTRAN 77 and migrated to Java. The case study consists of the following modules:

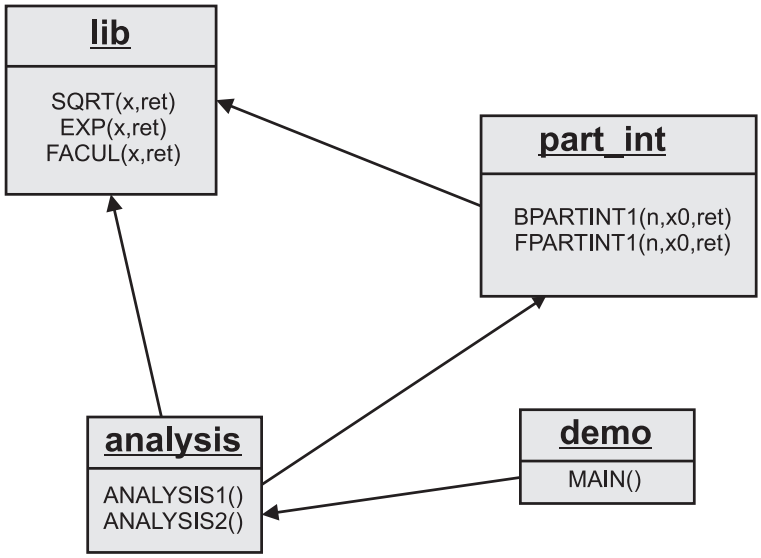

Figure 8.1: Relationship of Modules

- **demo**

  Contains the entry point (main() function) for the whole system.

- **analysis**

  Tests the algorithms and presents the results to the user. ANALYSIS1 does the partial integration while ANALYSIS2 demonstrates the square root function.

- **part_int**

  The actual partial integration algorithms are implemented in this module.

- **lib**

  All basic functions like $\sqrt{x}$, e^x or $x!$ are defined here.

The first step in the migration process is to translate the system into Weak Unsafe Typed WSL. Because the length of data types in C has never been defined clearly the translator has to know the exact length of each data type based on the used platform and compiler. Besides the lengths of data types the translation will also insert explicit type casts in every expression which involves variables with different types. Listing 8.1 and 8.2 give an example of a translated expression from C to Weak-Unsafe Typed WSL.

```
double FPARTINT1(double x, int n) {

  double a = 0;
  double expv = 0;
  double xp = 0;
  int p = 0;
  int i = 0;
  double ret1 = 0;
...
  a = (a - (p * (pow(x, xp) / expv)));
...
```

Listing 8.1: Code Snipped from C Source

```
PROC FPARTINT1(REAL*8::x, INTEGER*4::n VAR REAL*8::ret) ==
...
  VAR <REAL*8::a:=f0, REAL*8::expv:=f0, INTEGER*4::xp:=f0,
        INTEGER*4::p:=0, INTEGER*4::i:=0, REAL*8::ret1:=f0 >:
...
    a := a - {REAL*8} p * x ** {REAL*8} xp / expv;
```

Listing 8.2: Code Snipped from Corresponding WSL Source

It is crucially important to explicitly state the conversion operations during the process to highlight possible flaws. The next step in the migration process is to verify the type correctness of the translation which is done by the type checker. If a program is free from downcasts it can be

automatically verified as strongly typed. In this case study, however, the first type check revealed a flaw within the code. Listing 8.3 and 8.4 show the translation problem. The flaw is actually a combination of two problems. The first problem occurs when a call passes parameters to a procedure. FORTRAN uses normally the call-by-reference method while C uses always call-by-value if no pointer is involved. There are several ways to solve this e.g. to save and store the parameter value around each call or to declare the parameter variables as internal variable and initialise them with the actual parameter. In this migration the second alternative was choosen (`x` as local variable within the procedure and `v_x` as actual parameter variable) as it keeps added code to a minimum.

```
int f = 0;
...
ret3 = EXP(f);
...
double EXP(double x) {
...
```

Listing 8.3: Code Snipped from C Source

```
INTEGER*4 f
...
ret3 = LIB_EXP(f)
...
FUNCTION LIB_EXP(v_x)
REAL*8 v_x, x
x = v_x
...
```

Listing 8.4: Code Snipped from Corresponding Wrongly Translated FORTRAN Source

Unfortunately, this solution would not fully resolve the code flaw. The second problem is more subtle and existed because of an implicit type cast. The method EXP has a double parameter while the parameter in the call is an integer. In C this is not a problem as the compiler implicitly inserts a type cast which puts the value of the integer into the double parameter of the function. In FORTRAN, however, this does not happen. Here the address of the integer would be passed to the procedure and the content would be interpreted as double. The wrong value is then copied to the local variable and will eventually falsify the whole result. Such errors are very hard to track because even newer FORTRAN compiler, like the GNU FORTRAN 4.2, do not give a single

warning in these situations. Fortunately, the FermaT Type System Editor detected this and the problem was solved by creating a new variable with the appropriate type in the calling procedure and an assignment to this variable with the required value before the call (see Listing 8.5).

```
VAR <INTEGER*4 :: f := 0, ..., REAL*8 :: r_f := f0, ... >:
...
r_f := {REAL*8} f;
EXP( r_f VAR ret3);
...
PROC EXP( REAL*8 :: x VAR REAL*8 :: ret) ==
...
```

Listing 8.5: Code Snipped from WSL Source

After this modification the code was successfully verified as strong and safe typed which made it possible to transform the legacy system into an object oriented system which started by clustering the procedures with the object identification algorithm into classes. As described before the algorithm utilises for this the call graph. In the case study the Fan-Out threshold for object identification was set to *2* which identified the methods ANALYSIS, ANALYSIS1 and ANALYSIS2 to be put as main method into an own class (see figure 8.2). The Class Cluster Depth was set

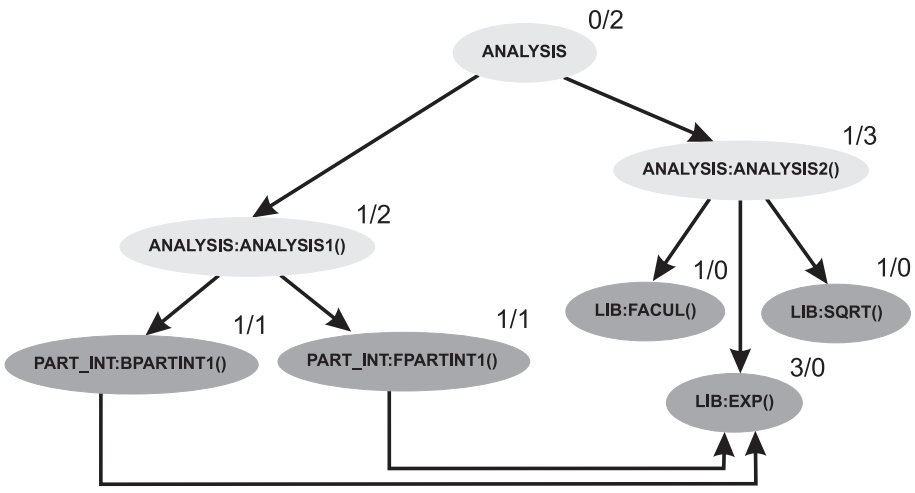

Figure 8.2: Object Identification

to *1* and the classes with the most amount of outgoing calls where selected first for object identification which caused the algorithm to include PART_INT1 and PART_INT2 into ANALYSIS1 and FACUL, SQRT and EXP into ANALYSIS2. The class with ANALYSIS, however, did not get any other procedure as there were no procedures left which were not already assigned to a

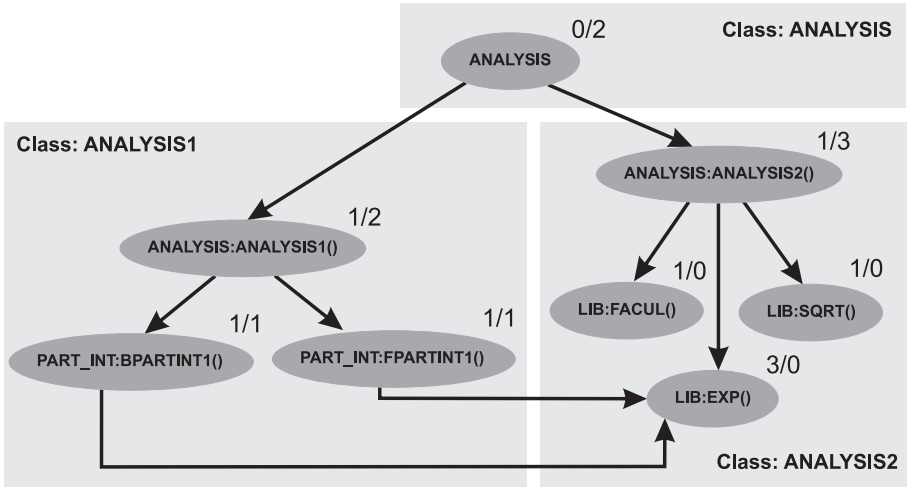

Figure 8.3: Object Identification

class. It should be noted that the chosen values for the parameters is purely based on empirical experience. The result of the algorithm is shown in figure 8.3. Listing 8.6 shows the resulting WSL OSTRUCTs which were generated to model the identified objects. Global variables like MAX_IT which were only used in one class (here ANALYSIS2) became local attributes of that class while global variables which were used in more than one class stayed in the global scope (declared as a PUBLIC and STATIC). Similar to the localised variables also all procedures which were only called within a class were declared as PRIVATE.

```
VAR < INTEGER*4 :: debug := 0,

 OSTRUCT :: ANALYSIS := <
           PUBLIC PROC ANALYSIS( )
 >,
 ANALYSIS :: OANALYSIS := < >,

 OSTRUCT :: ANALYSIS1 := <
     PUBLIC PROC ANALYSIS1( REAL*8, INTEGER*4 ),
     PRIVATE PROC BPARTINT1( REAL*8, INTEGER*4 VAR REAL*8 ),
     PRIVATE PROC FPARTINT1( REAL*8, INTEGER*4 VAR REAL*8 ),
 >,
 ANALYSIS1 :: OANALYSIS1 := < >,

 OSTRUCT :: ANALYSIS2 := <
     PRIVATE INTEGER*4 :: MAX_IT := 100,
```

```
      PUBLIC PROC ANALYSIS2( INTEGER*4 ),
      PRIVATE PROC FACUL( REAL*8 VAR REAL*8 ),
      PRIVATE PROC SQRT( REAL*8 VAR REAL*8 )
      PUBLIC PROC EXP( REAL*8 VAR REAL*8 )
 >,
 ANALYSIS2 :: OANALYSIS2 := < >
>:
```

Listing 8.6: Code Snipped from WSL Source

The last modification was done after the initial object identification by applying the type transformation "Reduce Inter-Class Relations" which counted the fan-in of every public method to move it to another class if the reviewed method is more called from an external class than from its current class. After application, the transformation found that EXP is called twice from ANALYSIS1 but only once from ANALYSIS2 and, as a result, moved the procedure to ANALYSIS1. The class diagram[2] of the translation result to Java can be seen in figure 8.4.

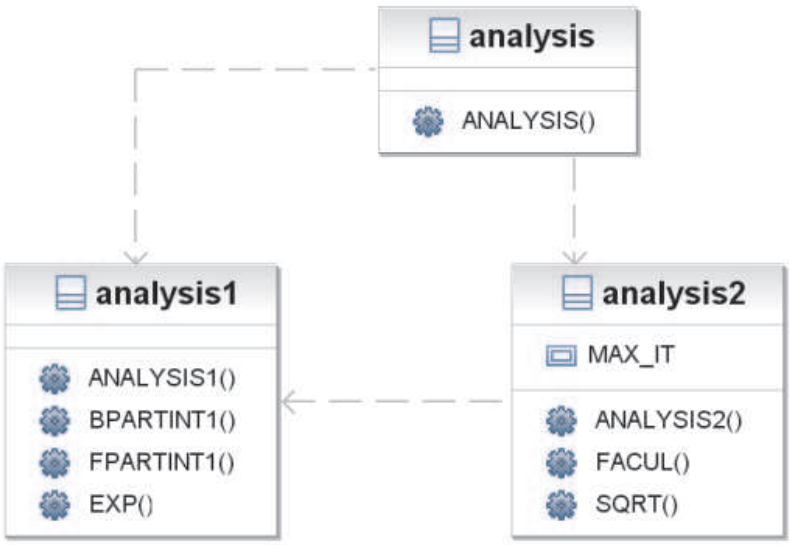

Figure 8.4: UML Diagram of Case Study 1

[2]Diagram was generated using eclipse with omondo UML plugin

8.3 Case Study 2

The second case study is to demonstrates the robustness and scalability of the approach by using a medium scale software system consisting of about 1600 lines of C code which will be migrated into Fortran 95. The example is a prototype backtracking program to find "relation-preserving mappings", as described in [PW94]. The case study consists of 8 modules with an average of 200 lines per module. The system contains 29 procedures which operate on 43 global variables. The migration was carried out in five steps:

1. Translation from C to Weak Unsafe Typed WSL.
2. Transformation from Weak Unsafe Typed WSL to Strong Unsafe Typed WSL.
3. Transformation from Strong Unsafe Typed WSL to Strong Safe Typed WSL.
4. Transformation from Strong Safe Typed WSL to Object Oriented Typed WSL.
5. Translation from Object Oriented Typed WSL to Fortran 95.

The first translation step involved besides the pure code to code translation also the insertion of explicit type casts whenever a type cast was done implicitly. Both translations (during step one and five) used the following data type conversions:

C	Weak Unsafe Typed WSL	Strong Unsafe / Safe / OO Typed WSL	FORTRAN 95
short	`INTEGER*2`	`INTEGER*2`	integer(2)
int	`INTEGER*2`	`INTEGER*4`	integer(4)
int[3]	`INTEGER*2`	`BOOLEAN*0`	logical
long	`INTEGER*2`	`INTEGER*4`	integer(4)
double	`REAL*8`	`REAL*8`	real(8)
short *	`LIST[?,?]<INTEGER*2>`	`LIST[?,?]<INTEGER*2>`	integer(2), dimension(?:?)

[3]This translation was only possible if the type inferencer concluded that the INTEGER variable was used as BOOLEAN in the C code.

int *	LIST[?,?]<INTEGER*4>	LIST[?,?]<INTEGER*4>	integer(4), dimension(?:?)
double *	LIST[?,?]<REAL*8>	LIST[?,?]<REAL*8>	real(8), dimension(?:?)
short ***	LIST[?,?]<LIST[?,?]< LIST[?,?]<INTEGER*2>>>	LIST[?,?]<LIST[?,?]< LIST[?,?]<INTEGER*2>{>}>	integer(2), dimension(?:?,?:?,?:?)

Table 8.1: Data Type Conversions of Case Study 2

The transition into Strong Unsafe Typed WSL during the second step of the migration involved the rewriting of some variables. The type inferencer was used to identify INTEGER variables which are used as BOOLEANs in C (Fortran 95 is able to express those variables with its data type logical):

```
int BUSH;
...
  BUSH = read_num(f);
...
  if (BUSH) {
    do_bush ();
  }
...
```

Listing 8.7: Code Snipped from C Source

```
VAR <
  ...
  BOOLEAN :: BUSH := TRUE
  ...
>:
...
  read_num(f VAR BUSH);
...
  IF BUSH THEN
    do_bush()
  FI;
...
```

Listing 8.8: Corresponding Code Snipped from WSL Source

The manual work during the whole migration process was done during the third step of the migration. The transition from Strong Unsafe Typed WSL to Strong Safe Typed WSL required that all C pointers had to be rewritten by other constructs. Fortunately, all pointers in the case study were used for array fields which could be expressed through the LIST type of WSL. However, their dimensions had to be gathered from `malloc` statements and manually inserted into the code. The following code example shows such a manual translation. The case study uses, among other arrays, four arrays with three dimensions. Each cell of these arrays allocates 2 bytes. The allocation of their memory is done in a special routine called "allocate_3D":

```
short ***RA, ***rel_to_A, ***rel_A_to, ***RB;
...
short*** allocate_3D (int x, int y, int z)
{
  int i, j;
  short ***result;
  /* allocate memory for a 3D short array of size 0..x by 0..y
   * by 0..z returns pointer to the array:
   */
  result = (short ***) malloc ((unsigned) ((x + 1)
    * sizeof (*result)));
  for (i = 0; i <= x; i++) {
    result[i] = (short **) malloc ((unsigned) ((y + 1)
      * sizeof (**result)));
    for (j = 0; j <= y; j++) {
      result[i][j] = (short *) malloc ((unsigned) ((z + 1)
        * sizeof (short)));
    }
  }
  return (result);
}
...
RA = allocate_3D (R, SA, SA);
rel_to_A = allocate_3D (R, SA, SA);
rel_A_to = allocate_3D (R, SA, SA);
RB = allocate_3D (R, SB, SB);
...
```

Listing 8.9: Code Snipped from C Source

All these manual allocations can be expressed within WSL with a simple variable declaration:

```
VAR <
  LIST[0:R]<LIST[0:SA]<LIST[0:SA]<INTEGER*2>>> RA := < >
  LIST[0:R]<LIST[0:SA]<LIST[0:SA]<INTEGER*2>>> rel_to_A := < >
  LIST[0:R]<LIST[0:SA]<LIST[0:SA]<INTEGER*2>>> rel_A_to := < >
  LIST[0:R]<LIST[0:SB]<LIST[0:SB]<INTEGER*2>>> RB := < >
  ...
> :
```

Listing 8.10: Corresponding Code Snipped from WSL Source

As FORTRAN supports since version 90 also object oriented structures, it was possible to use the Object Oriented layer of the Wide Spectrum Type System and change the paradigm from procedural to object oriented. The object identification algorithm was used to identify potential classes and possibilities for encapsulation. The call graph with the 29 procedures which was used for the object identification can be seen in figure 8.5. Procedures which will become the main method of a class and their outgoing calls are colored and their total number of outgoing calls is noted above their top right corner. The first fully automatic object identification used the following parameters:

- **Fan-Out Threshold** 7
- **Class Cluster Depth** 2
- **Class Identification Order** smallest number first

The resulting object oriented system can be seen in figure 8.6. A new class is created for every procedure whose total number of outgoing calls is greater or equal 7 (Fan-Out Threshold) while the procedure itself is placed inside the class as its first method. The Class Identification Order defines thereby which methods are first used for class identification. In this example the procedures with the smallest amount of outgoing calls were used first which gives the collection order: `find_good_element`, `do_bush`, `find_good_insert`, `do_pre_analysis`, `do_analyse`, `main`, `read_param`. In the second step the algorithm follows the call graph and includes also every procedure which is called by the identified procedure within 2 (Class Cluster Depth) steps also as class methods. Having a small Class Cluster Depth and using the procedures with the smallest amount of outgoing calls first gives usually a well balanced result since the first used procedures do not collect too many other procedures (as they have only a small number of outgoing calls) while the

last used procedures with the highest number of outgoing calls do not have much left to collect (since only procedures are moved into a class which are not already part of another class). During the first fully automated object identification step the system was able to identify 7 classes and distribute 23 procedures (out of 29) among them. The remaining 6 procedures were declared as public static and were put into a special class called global which models the global scope of the system. Many class methods (8 out of 23) could be declared as PRIVATE which indicates that the method is only called from the inside of a defined class. The majority of calls occur now within the classes which makes it much easier to follow the processing of the whole system. Around 37% (16 out of 43) of the global variables were localised among the classes which indicates that they are only affected by the methods of the class. After the object identification was carried out, the resulting object oriented WSL code was translated into FORTRAN 95. However, it would have been possible to further restructure the code by applying type transformations or by deleting dead code (procedures which are not called like `new_analyse`, `print_map` or `relpres`).

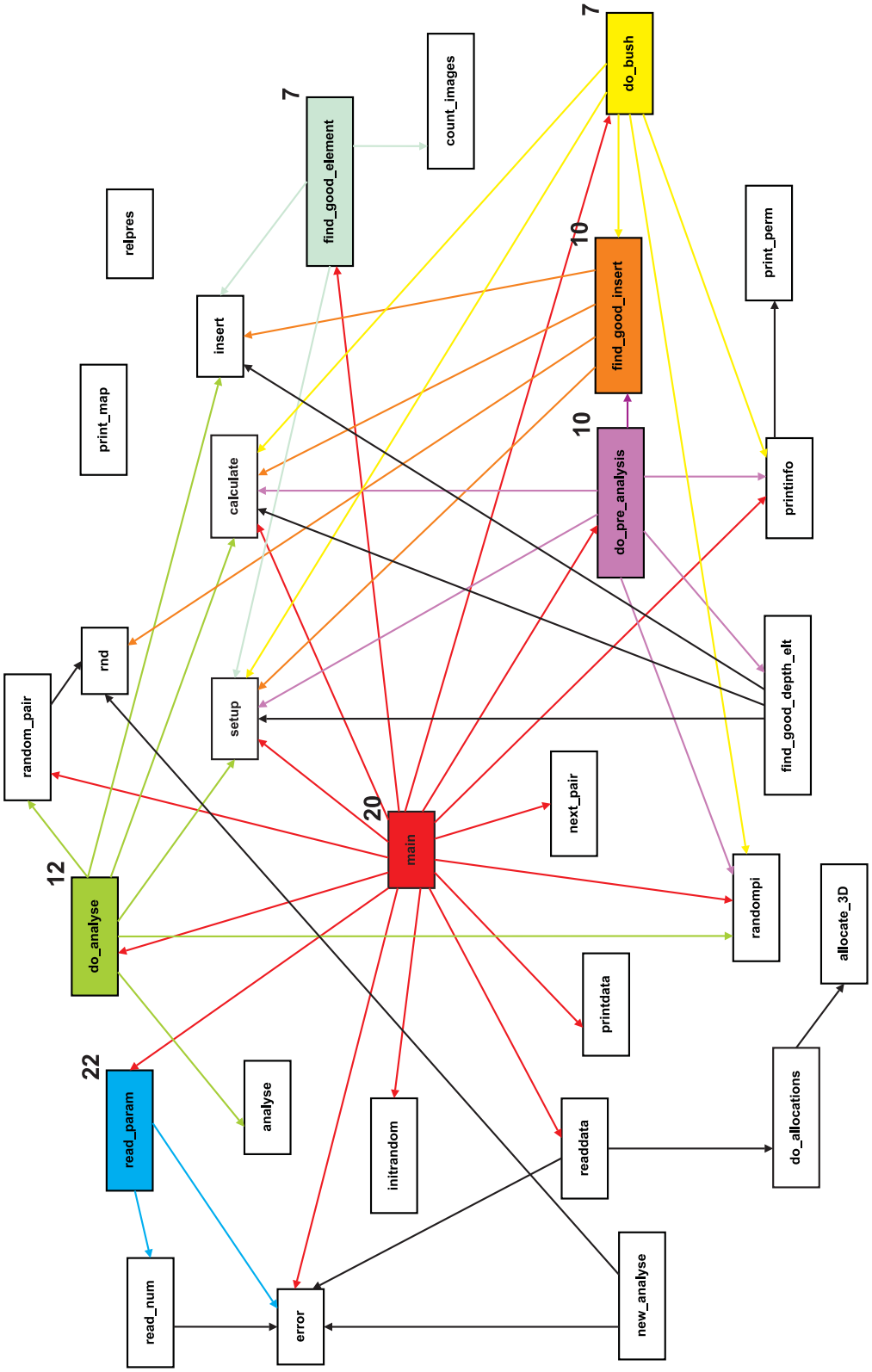

Figure 8.5: Call Graph of Case Study 2

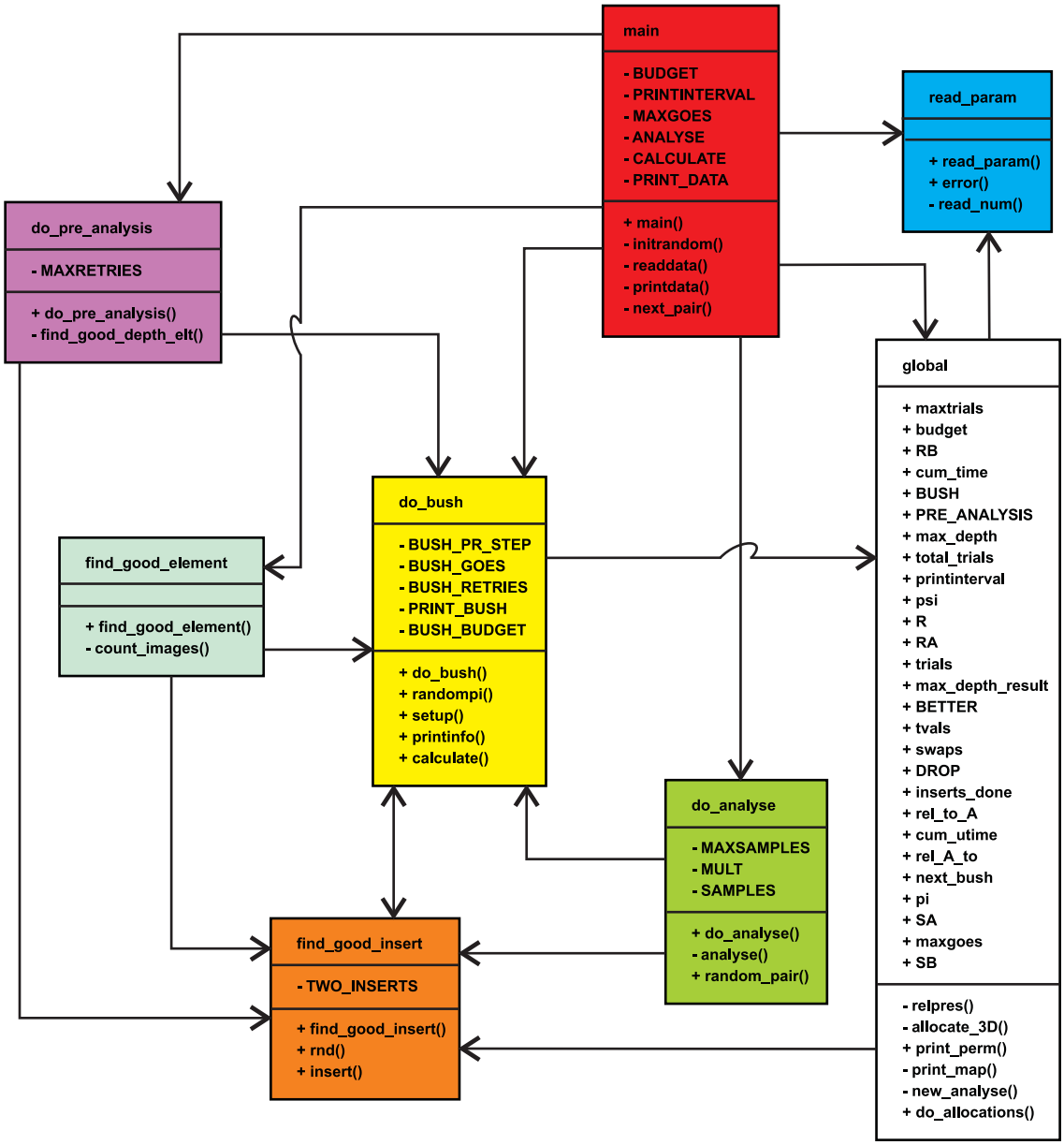

Figure 8.6: UML Diagram of Case Study 2

8.4 Summary

This chapter presented two case studies to show some practical issues of software migration and why a sophisticated type system is of utmost importance. The first case study showed the abilities of the type checker and explained how a migration can accidentally introduce errors due to implicit type

casting. Both case studies explained in detail the processing of the object identification algorithm and how it can be used to construct an object oriented system from a procedural oriented system. The general principle can be seen in case study 1 while case study 2 shows the application to a medium application. Having already an organised object oriented system with objects, which encapsulate already certain procedures and variables, it become significantly easier for a maintainer to understand and migrate the system successfully. The approach of using type transformations, which give an error message if they are not applicable, assures also that no new errors are introduced accidentally during the restructuring process. The second case study in particular demonstrated the robustness and scalability of the whole approach by migrating a medium scale software from C to Fortran 95. One of the most important steps in this case study was the correct identification of data types and their representation in the target language. Especially, the representation of pointers is critical as most strongly typed programming languages avoid explicit pointers. However, in many cases, if not a specific memory area has to be addressed, it is indeed possible to replace a pointer with a safe abstract data structure. The case studies show that WSL with the Wide Spectrum Type System is well suited for many different software migration tasks as it can precisely capture the semantics of many common programming languages which are used in current legacy systems.

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Mathison Turing

Chapter 9

Conclusion and Future Research

Objectives

- To evaluate and summarise the presented work.
- To state the limitation of the approach.
- To draw conclusions.
- To propose future work.

9.1 Summary of the Thesis

The thesis discussed how to augment the safety of program migration through a Wide Spectrum Type System. The presented approach specifically addresses data structure related issues of software migration through an intermediate language. The approach is unique as most of the current migration approaches use an abstract model (e.g. UML, flowchart, etc.) for intermediate representation rather than a programming language and a type system with adjustable expressiveness and strictness would not make sence for any ordinary programming language used for development. Only for a language like WSL which is used to change the abstraction level of source code, a Wide Spectrum Type System is useful. The most important key feature of the approach is the layered structure which enables the type system to adjust its expressiveness. As WSL can represent

programs at various levels of abstraction also the proposed type system is able to adjust expressiveness depending on the individual project requirements. Many features of the Wide Spectrum Type System have been discussed in the course of this thesis:

- The type system is organised in different layers which differ in expressiveness and strictness.
- The type system is defined through typing rules and attribute grammars which provides a sound mathematical foundation. This definition supports also the layered structure as only the typing rules have to be exchanged during a transition between layers.
- Emphasis on explicit typing.
  - The type of every variable must be known.
  - The storage of a variable is written in its declaration.
  - All conversions between data types have to be explicitly stated with a type cast.
- Enforcement of the type system by a flexible and scalable type checking algorithm.
- Introduction of the type system through its type inferencing algorithm.
- A program can be moved between the layers of the type system through type transformations.
- The object oriented layer allows the migration to object oriented languages.
- The object identification facilitates a smooth transition from procedural to object oriented paradigm.
- The approach can be integrated into the current FermaT migration process.

These features have been carefully chosen during the research investigation to find a good balance between benefits and "costs" in terms of complexity and constraints. There are two main benefits which arise from such a definition for the FermaT migration process.

1. Some identified flaws of the migration process could be eliminated.
   - Transformations which target abstractions on data structures are now possible.
   - Hidden data conversions (implicit type casting) are now revealed. A customer can be warned if the input data from current business rules are beyond the specification.

- Pointers can now be expressed and in some cases converted (e.g. to arrays).
- Variables which are in a special data format can now be modelled through the definition of artificial data types.

2. With the Wide Spectrum Type System also the range of potential source and target languages has been extended. WSL is now able to target most strongly typed imperative and even some object oriented programming languages.

These benefits, of course, are not without certain "costs" which are increased complexity for the WSL definition and the migration process as well as certain constraints:

- The type of every variable must be known.
- The size of non-container variables should be stated in bytes as part of a variable declaration.
- The size of container variables should be stated by a range.
- Container types can only contain content of one type. This type must always be stated in the declaration.
- Conversions between data types which are not carried out through a conversion function, have to be explicitly stated by a type cast.
- The operands of any operator must be of the same type.
- The object oriented layer is far less expressive and flexible as other state-of-the-art object oriented languages.

9.2 Limitations

As mentioned in the previous section, the research investigation had to find a good balance between benefits and "costs". This implies that the approach has also limitations and weaknesses:

- Sometimes it is not possible to transform a given typed program onto a higher type system layer. Especially excessive pointer usage and utilisation of weak typing in terms of dubious type conversions can make it impossible to proceed.

- The range of possible source and target languages is limited to procedural programming languages which use common data types and common typing techniques. Some programming languages with special features (which mostly involve dynamic typing) are not possible to migrate with this approach.
- Functional programming languages cannot be used as source or target language at all.
- The implementation of this approach is still in the prototype stage. The type checker may point out some false typing error due to wrongly formulated typing rules or bugs in the source code of the type checker.

9.3 Conclusions and Future Directions

The presented discussions and results in this thesis gave insight into problems which occur in current migration processes. Many of them are type related and can be tackled by the proposed Wide Spectrum Type System. The prototype tool and the case studies support this statement with concrete numbers and facts. Besides these concrete outcomes, the following facts have also become evident during the investigation:

- A sound, not necessarily excessive, formal foundation is essential for any reliable software engineering and reengineering approach.
- A strong type system and its strict enforcement significantly enhances the reliability of software.
- Flexibility, strictness and scalability are not necessarily mutually exclusive. A careful balance can incorporate all these attributes within a single approach.

The next step in the development of this research is to enhance the FermaT reengineering process to create a generic program transformation tool which is usable for most procedural languages. The development will be realised in three successive work packages (WP):

WP1 - Type enhancement

- Architectural justification of the FermaT reengineering process to use typed WSL.
- Integration of type inference and a type checking into the FermaT transformation system.
- Extending the transformation bank of FermaT with type transformations.
- Integration of the object identification process.

WP2 - Language translator development

- Development of a generic translator from procedural languages to WSL.
- Implementation of translators for C, Cobol and Java.

WP3 - Engineering of the software evolution process

- Integration of the ideas into a reengineering process.
- Evaluation of the process by using various case studies from industry.
- Deployment of the process to SML.

After this the research may be continued in a number of ways. Possible future directions for this approach are:

- The object identification algorithm can be refined further. The set of transformations for the object oriented layer can be extended by developing completely new transformations or by combining existing transformations into compositional transformations. The result will be a powerful object oriented refactoring system.
- The layered structure of the type system may be enhanced by defining additional layers. For example layers which target languages such as C++ which are object oriented but with a weak typing.
- Enriching the type system to do data refinement. Introduction of abstract data types which can be used to migrate from a concrete data type to an abstract and from abstract to a new concrete type.

Bibliography

[ACPP91] **M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin**, *Dynamic Typing in a Statically Typed Language*, ACM Transactions on Programming Languages and Systems, volume 13(2): pp. 237–268 (1991).

[All81] **F. E. Allen**, *The History of Language Processor Technology in IBM*, IBM Journal of Research and Development, volume 25(5): pp. 535–548 (1981).

[Ars79] **J. J. Arsac**, *Syntactic source to source transforms and program manipulation*, Communications of the ACM, volume 22(1): pp. 43–54 (1979).

[Bac60] **J. W. Backus**, *Report on The Algorithmic Language ALGOL 60*, Communications of the ACM, volume 2: pp. 299–314 (1960).

[Bac88] **C. Bachmann**, *A CASE for Reverse Engineering*, Datamation (July 1, 1988).

[Bac98] **J. Backus**, *The History of Fortran I, II, and III*, IEEE Annals of the History of Computing, volume 20(4): pp. 68–78 (1998).

[Bak90] **H. G. Baker**, *The Nimble Type Inferencer for Common Lisp-84*, Pre-publication draft, Nimble Computer Corp. (1990), submitted to ACM TOPLAS.

[BFK+94] **P. Baumann, J. Fässler, M. Kiser, Z. Öztürk, and L. Richter**, *Semantics-based Reverse Engineering*, Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland (1994).

[Bro75] **F. P. Brooks**, *The Mythical Man Month: Essays on Software Engineering* (Addison-Wesley, 1975), first edition.

[Bro87] **F. P. Brooks**, *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, volume 20(4): pp. 10–19 (1987).

[Byo98] **J. Byous**, *Java technology: The early years.*, Sun Developer Network (1998).

[Car91] **L. Cardelli**, *Typeful Programming*, in *Neuhold & Paul (Eds.), Formal Description of Programming Concepts, Springer-Verlag* (1991).

[Car04] **L. Cardelli**, *Type systems*, The Computer Science and Engineering Handbook, CRC Press (2004).

[Chu36] **A. Church**, *An unsolvable problem of elementary number theory*, American Journal of Mathematics, volume 58: pp. 345–363 (1936).

[Chu40] **A. Church**, *A Formulation of the Simple Theory of Types*, The Journal of Symbolic Logic, volume 5: pp. 56–68 (1940).

[CW85] **L. Cardelli and P. Wegner**, *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, volume 17(4): pp. 471–522 (1985).

[CW96] **E. M. Clarke and J. M. Wing**, *Formal methods: state of the art and future directions*, ACM Computing Surveys, volume 28(4): pp. 626–643 (1996).

[Dij72] **E. W. Dijkstra**, *The Humble Programmer*, Commun. ACM, volume 15(10): pp. 859–866 (1972).

[Dij76] **E. W. Dijkstra**, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[Dug07] **J. Duggan**, *Assessing the Age of Software Languages and Tools*, Gartner ras core research note, Gartner Inc. (2007).

[Eur96] **European Space Agency**, *Ariane 5: Flight 501 Failure. Report by the Enquiry Board*, Technical report (1996).

[FF95] **Y. A. Feldman and D. A. Friedman**, *Portability by Automatic Translation - A Large-Scale Case Study*, in *Proceedings: 10th Knowledge-Based Software Engineering Conference* (IEEE Computer Society Press, 1995), pp. 123–130, ISBN 0-8186-7204-8, 0-8186-7206-4, ISSN 1068-3062.

[GK95] **H. Gall and R. Klösch**, *Finding Objects in Procedural Programs: An Alternative Approach*, in **L. Wills, P. Newcomb, and E. Chikofsky** (editors), *Proceedings: Second Working Conference on Reverse Engineering* (IEEE Computer Society Press, 1995), pp. 208–216, ISBN 0-8186-7111-4.

[GM95] **J. Gosling and H. McGilton**, *The Java Language Environment. A White Paper* (sun, 1995).

[Gos97] **J. Gosling**, *The Feel of Java*, IEEE Computer, volume 30(6): pp. 53–58 (1997).

[Hin01] **M. Hind**, *Pointer analysis: haven't we solved this problem yet?*, in *PASTE* (ACM, 2001), pp. 54–61, ISBN 1-58113-413-4.

[Hol94] **J. R. Holmevik**, *Compiling SIMULA: A Historical Study of Technological Genesis*, IEEE Annals of the History of Computing, volume 16(4): pp. 25–?? (1994), ISSN 1058-6180.

[Hud89] **P. Hudak**, *Conception, evolution and application of functional programming languages*, Comp. Surveys, volume 21(3): pp. 359–411 (1989).

[IEE85] **IEEE Computer Society**, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985* (Institute of Electrical and Electronics Engineers, New York, 1985), reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[Int56] **International Business Machines Corporation**, *The FORTRAN automatic coding system for the IBM 704 EDPM, Programmers Reference Manual* (1956).

[Int94] **International Business Machines Corporation**, *IBM COBOL/400 User's Guide* (1994).

[Int96] **International Organization for Standardization**, *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF* (International Organization for Standardization, pub-ISO:adr, 1996).

[Int05] **International Business Machines Corporation**, *IBM XL C/C++ Enterprise Edition V8.0 for AIX Programming Guide* (2005).

[JG93] **G. L. S. Jr. and R. P. Gabriel**, *The Evolution of Lisp*, in *HOPL Preprints* (1993), pp. 231–270.

[Jon96] **T. C. Jones**, *Programming Languages Table*, Software Productivity Research Inc., Release 8.2 (1996).

[Jon98] **T. C. Jones**, *The Year 2000 Software Problem* (Addison-Wesley, 1998).

[Joy92] **I. Joyner**, *C++?? – A Critique of C++*, Technical report, UNISYS-ACUS (1992).

[Kar74] **C. R. Karp**, *Languages with Expressions of Infinite Length* (North-Holland, Amsterdam, 1974).

[Kle35] **S. Kleene**, *A theory of positive integers in formal logic*, American Journal of Mathematics, volume 57: pp. 153–173 and 219–244 (1935).

[Knu68] **D. E. Knuth**, *Semantics for context-free languages*, Mathematical Systems Theory 2, pp. 127–145 (1968).

[Knu90] **D. E. Knuth**, *The Genesis of Attribute Grammars*, in *INRIA: Attribute Grammars and their Applications, International Conference (WAGA)* (Springer, Berlin - Heidelberg - New York, 1990), pp. 1–12, ISBN 3-540-53101-7.

[KR78] **B. W. Kernighan and D. M. Ritchie**, *The C Programming Language* (Prentice Hall, Inc., "Englewood Cliffs, NJ", 1978), first edition.

[KR88] **B. W. Kernighan and D. M. Ritchie**, *The C Programming Language* (Prentice Hall, Inc., 1988), second edition.

[Lad06] **M. Ladkau**, *FermaT-UML An UML-intensive tool for assembler comprehension*, Diplomarbeit, Fachhochschule Oldenburg, Ostfriesland, Willhelmshaven (2006).

[Lan65] **P. J. Landin**, *Correspondence between ALGOL 60 and Church's Lambda-notation: part I*, Communications of the ACM, volume 8 , Issue 2: pp. 89–101 (February 1965).

[Lan97] **G. L. Lann**, *An analysis of the Ariane 5 flight 501 failure - A system engineering perspective*, in *ECBS* (IEEE Computer Society, 1997), pp. 339–246.

[LBSB80] **B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton**, *Software Maitenance Management* (Addison-Wesley, Reading, 1980), ISBN 0-201-04205-3.

[Leh96] **M. M. Lehman**, *Laws of Software Evolution Revisited*, in *European Workshop on Software Process Technology* (Springer, Berlin, 1996), pp. 108–124.

[Li07] **S. Li**, *A Program Transformation Step Prediction based Reengineering Approach*, Phd thesis, De Montfort University (2007).

[LP99] **L. Lamport and L. C. Paulson**, *Should Your Specification Language Be Typed?*, ACMTOPLAS: ACM Transactions on Programming Languages and Systems, volume 21 (1999).

[Man08] **P. Manchester**, *COBOL thwarts California's Governator*, The Register (2008).

[Mas06] **J. R. Mashey**, *The long road to 64 bits*, ACM Queue: Tomorrow's Computing Today, volume 4(8): pp. 24–35 (2006).

[Mat01] **Y. Matsumoto**, *Ruby in a Nutshell* (O'Reilly, 2001), ISBN 0596002149.

[McC60] **J. McCarthy**, *Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I*, Communications of the ACM, volume 3(4): pp. 184–195 (1960).

[McC78] **J. McCarthy**, *History of LISP*, SIGPLAN Notices, volume 13(8) (1978).

[Mey97] **B. Meyer**, *Object-Oriented Software Construction* (Prentice-Hall, Englewood Cliffs, 1997), second edition.

[Mil04] **R. C. Millham**, *Evolution of Batch-Oriented COBOL Systems into Object-Oriented Systems through Unified Modelling Language*, Phd thesis, De Montfort University (2004).

[Mit06] **R. L. Mitchell**, *Cobol: Not Dead Yet*, Computer World (2006).

[Mor94] **C. Morgan**, *Programming from Specifications* (Prentice-Hall, Englewood Cliffs, NJ, 1994), second edition.

[Moy92] **P. J. Moylan**, *The Case against C*, Technical Report TR–EE9240, Centre for Industrial Control Science, Department of Electrical and Computer Engineering, University of Newcastle, N.S.W. 2308, Australia (1992).

[MR87] **C. Morgan and K. Robinson**, *Specification Statements and Refinement*, IBM J. Res. Dev., volume 31(5): pp. 49–68 (1987).

[NR69] **P. Naur and B. Randell** (editors), *Software Engineering: Report on a conference sponsored by the NATO Science Committee* (NATO Scientific Affairs Division, 1969).

[Par94] **D. L. Parnas**, *Software Aging*, in *Proceedings of the 16th International Conference on Software Engineering* (IEEE Computer Society Press, 1994), pp. 279–287.

[Pie02] **B. C. Pierce**, *Types and Programming Languages* (The MIT Press, Cambridge, Massachusetts, 2002).

[PP92] **T. Pittman and J. Peters**, *The Art of Compiler Design: Theory and Practice* (Prentice-Hall, 1992), ISBN 0-13-046160-1.

[PS91] **J. Palsberg and M. I. Schwartzbach**, *Object-Oriented Type Inference*, in **N. Meyrowitz** (editor), *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (ACM Press, New York, NY, 1991), volume 26.

[PS94] **J. Palsberg and M. I. Schwartzbach**, *Object-Oriented Type Systems* (Wiley Professional Computing, 1994).

[PW94] **H. A. Priestley and M. P. Ward**, *A Multipurpose Backtracking Algorithm*, Journal of Symbolic Computation, volume 18(1): pp. 1–40 (1994).

[PZL98] **S. Pidaparthi, H. Zedan, and P. Luker**, *Conceptual Foundations for the Design Transformation of Procedural Software to Object-Oriented Architecture* (1998).

[Sne92] **H. M. Sneed**, *Migration of procedurally oriented COBOL programs in an object-oriented architecture*, IEEE Conference on Software Maintenance, pp. 105–116 (1992).

[Sne00] **H. M. Sneed**, *Encapsulation of legacy software: A technique for reusing legacy software components*, Annals Software Engineering, volume 9: pp. 293–313 (2000).

[Sto85] **B. Stoustrup**, *The C++ Programming Language* (Addison-Wesley, 1985).

[SV01] **H. M. Sneed and C. Verhoef**, *Reengineering the Corporation - A Manifesto for IT Evolution* (2001).

[Swa76] **E. B. Swanson**, *The Dimensions of Maintenance*, in *ICSE* (1976), pp. 492–497.

[TDPR+08] **M. Torchiano, M. Di Penta, F. Ricca, A. De Lucia, and F. Lanubile**, *Software migration projects in Italian industry: Preliminary results from a state of the practice*

survey, Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference, pp. 35–42 (2008).

[War89] **M. P. Ward**, *Proving Program Refinements and Transformations*, Dphil thesis, Oxford University (1989).

[War99] **M. P. Ward**, *Assembler to C Migration Using the FermaT Transformation System*, in *ICSM* (1999), pp. 67–76.

[War04] **M. P. Ward**, *Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations*, Science of Computer Programming, Special Issue on Program Transformation 52, pp. 213–255 (2004).

[WB95] **M. P. Ward and K. H. Bennett**, *Formal Methods for Legacy Systems*, Journal of Software Maintenance: Research and Practice, volume 7(3): pp. 203–220 (1995).

[WCM89] **M. P. Ward, F. W. Calliss, and M. Munro**, *The Maintainer's Assistant*, in *Proceedings of the International Conference on Software Maintenance*, IEEE (IEEE Computer Society Press, 1989), p. 307.

[Wei91] **M. Weiser**, *The computer for the 21st century*, Scientific American, volume 265(3): pp. 94–104 (1991).

[WH03] **M. P. Ward and T. Hardcastle**, *WSL Programmer's Reference Manual* (2003).

[Wir77] **N. Wirth**, *What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?*, Communications of the ACM, volume 20(11): pp. 822–3 (1977), ISSN 0001-0782.

[WZ05] **M. P. Ward and H. Zedan**, *MetaWSL and Meta-Transformations in the FermaT Transformation System*, in *COMPSAC* (IEEE Computer Society, 2005), pp. 233–238, ISBN 0-7695-2413-3.

[WZ06] **M. P. Ward and H. Zedan**, *Analysing and Abstracting Legacy Assembler Code via Conditioned Semantic Slicing*, Technical report, Software Technology Research Laboratory (2006).

[WZ07] **M. P. Ward and H. Zedan**, *Slicing as a Program Transformation*, ACM Transactions on Programming Languages and Systems, volume 29 (2007).

[WZH04] **M. P. Ward, H. Zedan, and T. Hardcastle**, *Legacy Assembler Reengineering and Migration*, in *ICSM* (IEEE Computer Society, 2004), pp. 157–166, ISBN 0-7695-2213-0.

[YW03] **H. Yang and M. P. Ward**, *Successful Evolution of Software Systems* (Artech house, INC., 2003).

Appendix A

Source Code

A.1 Type Checking / Type Inferencing Algorithm

Listing A.1: Type checking / Type Inferencing Algorithm in Java

```
package fme.components.typesystemeditor.wsl.inferencer.interfaces;

import java.util.Iterator;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

import fme.components.typesystemeditor.wsl.structures.TSEASTNode;
import fme.components.typesystemeditor.wsl.structures.TSETypeRuleTable;

public class WstsTypeCheckTypeInference {

    // The type rule table
    private static TSETypeRuleTable rt;

    // Flag to toggle additional debug messages
    private static boolean debug = false;

    // Flag to indicate that an error occured
    private static boolean error;

    // The log output of the algorithm
    private static StringBuffer log;

    public boolean isError() {
        return error;
    }

    public String inferLocation(TSEASTNode node) {
        String p[], c[], pre, con, type;
        Vector<Vector<String>> nodeRules;
        boolean cont = false;

        // If node has already a type or has been visited before return
        if (debug)
            System.out.println(node + "  Infer>" + node + "  type:"
                + node.getInternalValue());
        if (node.getInternalValue() != null) {
            if (debug)
                System.out.println(node + "    Node has already type "
                    + node.getInternalValue());
```

```
        return node.getInternalValue();
    }
    if (node.getInternalValue() != null
        && node.getInternalValue().equals("Pending....")) {
        if (debug)
            System.out
                .println("  Node is already part of inference process..."
                    + node.getInternalValue());
        return node.getInternalValue();
    } else {
        if (debug)
            System.out.println(node + "    Going on ...");
    }

    // Mark the node
    node.setInternalValue("Pending....");

    // Get the rules for the node
    nodeRules = rt.getRulesForConclusionNode(node.getSpecificType());
    if (nodeRules == null) {
        if (debug) {
            System.out.println(node + "    No rule can be found for node");

        }
        node.setInternalValue(null);
        return null;
    }

    // Here beginns now the inference / check algorithm
    // ================================================

    // Go through all rules
    if (debug)
        System.out.println(node + "    Rules:");

    // Go through all rules for this node
    for (int j = 0; j < nodeRules.size(); j++) {
        if (debug)
            System.out.println(node + "  Rule #>" + nodeRules.get(j).get(0)
                + " | " + nodeRules.get(j).get(1));
        pre = nodeRules.get(j).get(0);
        con = nodeRules.get(j).get(1);

        p = pre.split(";");
        c = con.split(";");
```

```
// Infer the node type directly if no premise is given in the rule
if (pre.equals("")) {
    if (debug)
        System.out.println(node + "    No Premises given...");
    type = c[0].split(":")[1];
    if (type.equals("VC0[VC1]")) {
        type = ((TSEASTNode) node.getChildAt(0)).getValue();
        if (node.getChildCount() == 2) {
            inferLocation((TSEASTNode) node.getChildAt(1));
            type = type
                + "<"
                + ((TSEASTNode) node.getChildAt(1))
                    .getInternalValue() + ">";
        }
        setType(type, node);
    } else if (node.getInternalValue() == null
        || node.getInternalValue().equals("Pending....")
        || node.getInternalValue().equals("VOID")) {
        if (type.endsWith(("#V")))
            type = type.replace("#V", "*" + node.getValue());
        setType(type, node);
    }
}

// Handle the parts of the rule that try to identify a node
for (int i = 0; i < p.length; i++) {
    if (p[i].startsWith("Parent=")) {
        String d[] = p[i].split("=");
        // Check if the partent should be of a certain type
        if (!node.getParent().getSpecificType().equals(d[1])) {
            if (debug)
                System.out.println(node + "    Parent isn't a "
                    + d[1] + " node...");
            cont = true;
            break;
        } else {
            if (debug)
                System.out.println(node + "    Parent is a " + d[1]
                    + " node...");
        }
    }
}
if (cont) {
    cont = false;
```

```
            continue;
        }

        // Handle the parts of the rule that mention child nodes
        for (int i = 0; i < p.length; i++) {
            if (p[i].startsWith("Child")) {
                if (handleChildren(p[i], c[0], node) == null)
                    return null;
            }
        }

        // Handle the parts of the rule that mention the parent node
        for (int i = 0; i < p.length; i++) {
            if (p[i].startsWith("Parent")) {
                if (node.getParent().getSpecificType().equals("T_Cast")) {
                    cont = true;
                    break;
                } else if (handleParent(p[i], c[0], node) == null)
                    return null;
            }
        }

        if (cont) {
            cont = false;
            continue;
        }

        // Handle conclusion
        if (c.length > 1)
            doConclusion(p, c, node);

        if (debug)
            System.out.println(node + "  Checking Children ...");
        // Check if a child can be inferred more precisely
        checkChildren(p, c, node);
    }

    // Try the parent node
    if (node.getParent().getInternalValue() == null) {
        if (debug)
            System.out.println(node + "    Now trying parent node of "
                + node + "...");
        inferLocation(node.getParent());
    }
```

```
        // Check if the parent needs revalidation
        if (debug)
            System.out.println(node + "  Checking Parent ...");
        checkParent(node.getParent());

        if (node.getInternalValue() != null
            && !node.getInternalValue().equals("Pending....")) {
            return node.getInternalValue();
        } else {
            node.setInternalValue(null);
            return null;
        }
    }

    // Subroutines for the inference algorithm
    // =======================================

    private void checkParent(TSEASTNode node) {
        String pre, p[], reftype;
        TSEASTNode child;
        Vector<Vector<String>> nodeRules;
        Vector<String> r;
        Iterator<Vector<String>> ir;
        Iterator<TSEASTNode> it;

        nodeRules = rt.getRulesForConclusionNode(node.getSpecificType());

        if (nodeRules == null)
            return;

        ir = nodeRules.iterator();

        while (ir.hasNext()) {
            r = ir.next();
            pre = r.get(0);
            p = pre.split(";");

            // Test if all children should be the same
            if (p[0].startsWith("Child(A)")) {
                // Test if all children are the same
                it = node.getChildren().iterator();
                reftype = null;
                while (it.hasNext()) {
                    child = it.next();
                    if (reftype == null)
```

```
                    reftype = child.getInternalValue();
                else if (child.getInternalValue() != null
                    && !typeEquals(reftype, child.getInternalValue())
                    && !reftype.equals("Pending....")) {
                    if (debug)
                        System.out.println(node
                            + "  Revalidating Parent ...");
                    node.setInternalValue(null);
                    inferLocation(node);
                }
            }
        }
    }
}

private void checkChildren(String[] premise, String[] conclusion,
    TSEASTNode node) {

    String pre, con, p[], c[];
    Vector<Vector<String>> nodeRules;
    Vector<String> r;
    TSEASTNode child;
    Iterator<TSEASTNode> it;
    Iterator<Vector<String>> ir;

    if (debug)
        System.out.println(node + "  Checking Node ...");

    for (int j = 0; j < premise.length; j++) {
        // Child must have the Parent in its premise
        if (premise[j].equals("Parent:T1") && conclusion[0].contains("T1")) {
            // Check if parent has information that child has not
            if (node.getParent().getInternalValue() != null
                && !node.getParent().getInternalValue().equals(
                    "Pending....") && node.getInternalValue() != null
                && node.getInternalValue().contains("VOID")) {
                // Child must have type VOID inside
                if (node.getInternalValue() != null) {
                    if (conclusion[0].contains("[T1]")) {
                        setType(
                            conclusion[0].split(":")[1]
                                .replace("[T1]", "<"
                                    + node.getParent().getInternalValue()
                                    + ">"), node);
                    } else if (conclusion[0].contains("T1")) {
```

```
                        setType(node.getParent().getInternalValue(), node);
                    }
                }
            }
        }

        if (debug)
            System.out.println(node + "  Checking Children ...");
        it = node.getChildren().iterator();
        while (it.hasNext()) {
            child = it.next();
            nodeRules = rt.getRulesForConclusionNode(child.getSpecificType());
            if (nodeRules != null) {
                ir = nodeRules.iterator();
                while (ir.hasNext()) {
                    r = ir.next();
                    pre = r.get(0);
                    con = r.get(1);
                    p = pre.split(";");
                    c = con.split(";");
                    checkChildren(p, c, child);
                }
            }
        }
    }

    private String setType(String type, TSEASTNode node) {
        if (debug)
            System.out.println(node + "  *** " + node + " -> " + type);
        if (type != null) {
            type = type.replace("[", "<");
            type = type.replace("]", ">");
        }
        node.setInternalValue(type);
        return type;
    }

    private String setNodeToType(String type, String premise,
        String conclusion, TSEASTNode node) {
        String p[], c[];

        p = premise.split(":");
        c = conclusion.split(":");
```

```
        // Set to type VOID if no type was inferred
        if (type == null || (type != null && type.equals("Pending....."))) {
            // Set content type void if the children are the contents of
            // a container node
            if (c[1].contains("[T1]"))
                type = c[1].replace("[T1]", "[VOID]");
            // Set the type to void
            else
                type = "VOID";
        }

        if (!c[1].contains("T1") && !c[1].contains("VOID")) {
            setType(c[1], node);
        } else if (c[1].equals("T1")) {
            if (p[1].contains("[T1]") && type.contains("<")) {
                setType(type
                    .substring(type.indexOf("<") + 1, type.length() - 1), node);
            } else {
                setType(type, node);
            }
        } else if (p[1].contains("[T1]") && c[1].contains("[T1]")) {
            setType(type, node);
        } else if (c[1].contains("[T1]")) {
            setType(c[1].replace("[T1]", "<" + type + ">"), node);
        } else {
            setType(type, node);
        }
        return "";
    }

    private String handleParent(String premise, String conclusion,
        TSEASTNode node) {

        String c[], type;

        c = conclusion.split(":");

        if (debug)
            System.out.println(node + "    For Parent...");

        type = inferLocation(node.getParent());

        if (node.getInternalValue() != null
            && !node.getInternalValue().equals("Pending....")) {
            if (debug)
```

```
            System.out.println(node + "    Parent has inferred the type "
                + node.getInternalValue() + " for node " + node);
        } else if (premise.equals("Parent:T1")) {
            if (c[1].equals("T1")) {
                setType(type, node);
            } else {
                Logger.getLogger(this.getClass().getCanonicalName()).log(
                    Level.WARNING,
                    "  Conclusion type :" + conclusion + " not implemented!");
                error = true;
            }
        } else {
            Logger.getLogger(this.getClass().getCanonicalName()).log(
                Level.WARNING,
                "  Premise type :" + premise + " not implemented!");
            error = true;
        }

        return "";
    }

    private String handleChildren(String premise, String conclusion,
        TSEASTNode node) {

        String p[];

        p = premise.split(":");

        // For all childs
        if (p[0].equals("Child(A)")) {
            return handleAllChildren(premise, conclusion, node);
        } else {
            return handleSingleChild(premise, conclusion, node);
        }
    }

    private String handleAllChildren(String premise, String conclusion,
        TSEASTNode node) {
        String p[], c[], type, reftype, ctype;
        TSEASTNode child;
        Iterator<TSEASTNode> it;

        p = premise.split(":");
        c = conclusion.split(":");
```

```
if (debug)
    System.out.println(node + "    For all Children...");

// The type of the children is requested
// (This means the children must have the same type)
if (p[1].contains("T1")) {

    if (debug)
        System.out.println(node + "    Infer type from Children");
    it = node.getChildren().iterator();

    reftype = null;
    type = null;

    // Check the types of the children
    while (it.hasNext()) {
        child = it.next();
        type = inferLocation(child);

        // Set the reference type
        if (reftype == null && type != null
            && !type.equals("Pending....")) {
            reftype = type;
        }

        // Set the reference type new if the type is more
        // specialised
        if ((reftype != null && type != null
            && reftype.contains("VOID") && !type.contains("VOID") && !type
            .equals("Pending...."))
            || (reftype != null && reftype.equals("VOID"))) {
            reftype = type;
        }

        // Detect if the children have different types
        if (reftype != null && type != null
            && !typeEquals(type, reftype) && !type.equals("VOID")
            && !type.equals("Pending....")) {

            if (type.contains("<VOID") && reftype.contains("<")
                && reftype.length() > type.indexOf("<VOID")) {
                if (!type.subSequence(0, type.indexOf("<VOID")).equals(
                    reftype.subSequence(0, type.indexOf("<VOID")))) {
                    if (debug)
                        System.out
```

```
                            .println("  Children have different types");
                    log.append("Children of node "
                        + node.getSpecificType() + " (row:"
                        + node.getRow() + ") have different types!\n");
                    error = true;
                    return null;
                }
            } else {
                if (debug)
                    System.out
                        .println("  Children have different types");
                log.append("Children of node " + node.getSpecificType()
                    + " (row:" + node.getRow()
                    + ") have different types!\n");
                error = true;
                return null;
            }
        }

        // Check if child should be of a specific container type
        if (p[1].contains("[")) {
            ctype = p[1].substring(0, p[1].indexOf("["));
            if (type != null && !type.startsWith(ctype)
                && !type.equals("VOID") && !type.equals("Pending....")) {
                if (debug)
                    System.out
                        .println("  Child should be of container type:"
                            + p[1]);
                log.append("Child of node " + node + " (row:"
                    + node.getRow() + ") should be of container type:"
                    + p[1] + "\n");
                error = true;
                return null;
            }
        }
    }

    setNodeToType(reftype, premise, conclusion, node);

    // Process all children
    // ====================

    // In case no child gave a result set to type VOID
    if (reftype == null
        || (reftype != null && reftype.equals("Pending...."))) {
```

```
            // Set content type void if the children are the contents of
            // a container node
            if (c[1].contains("[T1]"))
                reftype = c[1].replace("[T1]", "[VOID]");
            // Set the type to void
            else
                reftype = "VOID";
        }

        // The type in a container was inferred
        if (p[1].contains("[T1]") && !c[1].contains("[T1]")) {
            reftype = p[1].replace("[T1]", "[" + reftype + "]");
        }

        // Set the found type for all children
        if (debug)
            System.out.println(node + "    All Children have type:"
                + reftype);
        it = node.getChildren().iterator();
        while (it.hasNext()) {
            child = it.next();
            type = child.getInternalValue();
            if (type == null || type.equals("Pending....")
                || type.contains("VOID")) {
                setType(reftype, child);
            }
        }
    }
    // Set type of children
    else {
        if (debug)
            System.out.println(node + "    Set type of Children");

        it = node.getChildren().iterator();
        while (it.hasNext()) {
            child = it.next();

            // Infer Children from Child
            inferLocation(child);

            if (debug)
                System.out.println(node + "    Setting child " + child
                    + " to:" + p[1]);
            if (child.getInternalValue() != null
                && !child.getInternalValue().equals("Pending....")
```

```
                && !typeEquals(child.getInternalValue(), p[1])) {
                log.append("Child of node " + node.getSpecificType()
                    + " (row:" + node.getRow() + ") has type:"
                    + child.getInternalValue() + " but should have type:"
                    + p[1] + "\n");
            }
            setType(p[1], child);
        }
        setType(c[1], node);
    }

    return "";
}

private String handleSingleChild(String premise, String conclusion,
    TSEASTNode node) {
    int n;
    String p[], c[], type;
    TSEASTNode child;

    p = premise.split(":");
    c = conclusion.split(":");

    // Get the childs
    n = Integer.parseInt(p[0].substring(6, p[0].length() - 1));
    if (debug)
        System.out.println(node + "    For Child " + n + "...");

    // Infer type from child
    if (p[1].contains("T1")) {
        if (debug)
            System.out.println(node + "    Infer Type from Child");

        type = inferLocation(node.getChildren().get(n));

        setNodeToType(type, premise, conclusion, node);
    }
    // Set type of child
    else {
        if (debug)
            System.out.println(node + "    Set type of Child");

        child = node.getChildren().get(n);

        // Infer Children from Child
```

```
            inferLocation(child);

            if (debug)
                System.out.println(node + "    Setting child " + child + " to:"
                    + p[1]);
            setType(p[1], child);

            if (!c[1].contains("T1"))
                setType(c[1], node);
        }

        return "";
    }

    private String doConclusion(String[] premise, String[] conclusion,
        TSEASTNode node) {

        int n;
        String c[], type;
        TSEASTNode child;

        if (debug)
            System.out.println(node + "    Doing conclusion of node " + node
                + " ...");

        for (int i = 0; i < conclusion.length; i++) {

            c = conclusion[i].split(":");

            // Set a specific Child
            if (conclusion[i].startsWith("Child")) {
                n = Integer.parseInt(c[0].substring(6, c[0].length() - 1));

                if (debug)
                    System.out.println(node + "    For Child " + n + "...");

                child = node.getChildren().get(n);

                // Infer Children from Child
                inferLocation(child);

                type = node.getInternalValue();
                if (type.equals("Pending....")) {
                    type = "VOID";
                }
```

```
            if (!c[1].contains("T1")) {
                type = c[1];

            } else if (conclusion[0].contains("[T1]")
                && c[1].contains("[T1]")) {
                type = type.substring(type.indexOf("<") + 1,
                    type.length() - 1);
            } else if (c[1].contains("[T1]")) {
                type = c[1].replace("[T1]", "<" + type + ">");
            }

            // Only set type if child has a more general type
            if (type.contains("VOID")) {
                if (child.getInternalValue() == null
                    || child.getInternalValue().equals("Pending....")
                    || child.getInternalValue().contains("VOID"))
                    setType(type, child);
            } else {
                setType(type, child);
            }
        }
    }

    return "";
}

/**
 * Judges if two types are equal
 *
 * @param type1
 *            A type
 * @param type2
 *            Another type
 * @return True if the two types are the same
 */
private static boolean typeEquals(String type1, String type2) {
    boolean ret;

    if (type1.equals(type2))
        ret = true;
    else if (type1.contains("*") && !type2.contains("*")
        && type1.replaceFirst(".[0-9]+", "").equals(type2))
        ret = true;
    else if (type2.contains("*") && !type1.contains("*")
```

```
            && type2.replaceFirst(".[0-9]+", "").equals(type1))
            ret = true;
        else if (type1.replaceFirst(".[0-9]+", "").equals(
            type2.replaceFirst(".[0-9]+", ""))
            && (type1.contains("*0") || type2.contains("*0"))) {
            ret = true;
        } else
            ret = false;

        if (debug)
            System.out.println("Type Equal? " + type1 + " = " + type2 + " ->"
                + ret + "!");

        return ret;
    }
}
```

A.2 Case Study 1

A.2.1 Source Code in FORTRAN 77

```
      PROGRAM case_study

      IMPLICIT NONE

c     Global Variables

      INTEGER*4 debug
      COMMON debug

      INTEGER*4 LIB_MAX_IT
      COMMON LIB_MAX_IT

c Main Procedure

      REAL*8 x
      INTEGER*4 n

c     Global initialisation
      debug = 0
      LIB_MAX_IT = 100

      x = 1
      n = 10
      call ANALYSIS_ANALYSIS1(x,n)
      call ANALYSIS_ANALYSIS2(n)

      END PROGRAM case_study
```

Listing A.2: case_study.for

```
      SUBROUTINE ANALYSIS_ANALYSIS1(v_x,v_n)

      INTEGER*4 debug
      COMMON debug

      REAL*8 v_x, x
      INTEGER*4 v_n, n
```

```
REAL*8 PART_INT_BPARTINT1
REAL*8 PART_INT_FPARTINT1

REAL*8 ret1
x = v_x
n = v_n
ret1 = 0
write(*,*)"\nPartial Integration"
write(*,*)"------------------"
ret1 = PART_INT_BPARTINT1(x, n)
write(*,*)"Backward:",ret1
ret1 = PART_INT_FPARTINT1(x, n)
write(*,*)"Forward :",ret1

END

SUBROUTINE ANALYSIS_ANALYSIS2(v_n)

INTEGER*4 debug
COMMON debug

INTEGER*4 v_n, n

INTEGER*8 LIB_FACUL
REAL*8 LIB_SQRT
REAL*8 LIB_EXP

INTEGER*4 f
REAL*8 n1
REAL*8 n2
INTEGER*8 ret1
REAL*8 ret2
REAL*8 ret3
REAL*8 ret4
REAL*8 r_f
n = v_n
f = 0
r_f = 0
n1 = 0
n2 = 0
ret1 = 0
ret2 = 0
ret3 = 0
ret4 = 0
write(*,*)"\nANALYSIS2";
```

```
      write(*,*)"---------";
      DO f = 0, n, 1
        ret1 = LIB_FACUL(f)
        write(*,*)"Facul:",ret1
        n1 = ret1
        ret2 = LIB_SQRT(n1)
        write(*,*)"Sqrt :",ret2
        r_f = f
        ret3 = LIB_EXP(r_f)
        write(*,*)"Exp  :",ret3
        n2 = ret3
        ret4 = LIB_SQRT(n2)
        write(*,*)"Sqrt :",ret4
      ENDDO

      END
```

Listing A.3: analysis.for

```
c     Procedure:BPARTINT1
c     ===================
c     Partial Integration (backward) of t^n * e^-t to t
c     PARAM: x = Upper limit of Integration (lower limit is 0)
c     PARAM: n = power of t
c     RETURN: ret = Result of Integration

      FUNCTION PART_INT_BPARTINT1(v_x,v_n)

      INTEGER*4 debug
      COMMON debug

      REAL*8 PART_INT_BPARTINT1
      REAL*8 v_x, x
      INTEGER*4 v_n, n

      REAL*8 LIB_EXP

      REAL*8 a
      REAL*8 expv
      REAL*8 xp
      INTEGER*4 i
      REAL*8 ret1
      x = v_x
      n = v_n
      a = 0
```

```
      expv = 0
      xp = 0
      i = 0
      ret1 = 0
      ret1 = LIB_EXP(x)
      expv = ret1
      a = (1 - (1 / expv))
      DO i = 1, n, 1
        xp = x**i
        a = ((a * i) - (xp / expv))
        if (debug .eq. 1) THEN
          write(*,*) "BPARTINT1:",a
        ENDIF
      END DO
      PART_INT_BPARTINT1 = a

      END


c     Procedure:FPARTINT1
c     ===================
c     Partial Integration (forward) of t^n * e^-t to t
c     PARAM: x = Upper limit of Integration (lower limit is 0)
c     PARAM: n = power of t
c     RETURN: ret = Result of Integration

      FUNCTION PART_INT_FPARTINT1(v_x,v_n)

      INTEGER*4 debug
      COMMON debug

      REAL*8 PART_INT_FPARTINT1
      REAL*8 v_x, x
      INTEGER*4 v_n, n

      REAL*8 LIB_EXP

      REAL*8 a
      REAL*8 expv
      REAL*8 xp
      REAL*8 p
      INTEGER*4 i
      REAL*8 ret1
      x = v_x
      n = v_n
```

```
      a = 0
      expv = 0
      xp = 0
      p = 0
      i = 0
      ret1 = 0
      ret1 = LIB_EXP(x)
      expv = ret1
      a = (-x**n / expv)
      p = 1
      DO i = 0, (n-1), 1
        p = (p * (n - i))
        xp = (n - i - 1)
        a = (a - (p * (x**xp / expv)))
        if (debug .eq. 1) THEN
          write(*,*) "FPARTINT1:",a
        ENDIF
      END DO
      a = (a + p)
      PART_INT_FPARTINT1 = a

      END
```

Listing A.4: part_int.for

```
c     Procedure:SQRT
c     ==============
c     A quick square root function
c     PARAM: x = Number for square root
c     RETURN: ret = Square root of x

      FUNCTION LIB_SQRT(v_x)

      INTEGER*4 debug
      COMMON debug
      INTEGER*4 LIB_MAX_IT
      COMMON LIB_MAX_IT

      REAL*8 LIB_SQRT
      REAL*8 v_x, x

      REAL*8 n
      REAL*8 xnew
      INTEGER*4 i
      x = v_x
```

```
      n = (x / 2)
      xnew = 0
      i = 0
      DO i = 0, LIB_MAX_IT, 1
        xnew = ((n + (x / n)) / 2)
        IF (xnew .lt. x) THEN
          n = xnew
        ELSE
          EXIT
        ENDIF
        if (debug .eq. 1) THEN
          write(*,*) "SQRT:",xnew
        ENDIF
      END DO
      LIB_SQRT = xnew

      END


c     Procedure:EXP
c     =============
c     Calculates powers of Euler's number
c     PARAM: x = Power of e to calculate
c     RETURN: ret = Euler's number to the power of x

      FUNCTION LIB_EXP(v_x)

      INTEGER*4 debug
      COMMON debug

      REAL*8 LIB_EXP
      REAL*8 v_x, x

      INTEGER*4 i
      REAL*8 p
      REAL*8 s
      REAL*8 t
      x = v_x
      i = 1
      p = 1
      s = 1
      t = 2
      DO WHILE (s .ne. t)
        t = s
        p = ((p * x) / i);
```

```
        s = (s + p);
        i = (i + 1)
        if (debug .eq. 1) THEN
          write(*,*) "EXP:",s
        ENDIF
      END DO
      LIB_EXP = s

      END


c     Procedure:FACUL
c     =============
c     Calculates the factorial of x
c     PARAM: x = Factorial to calculate
c     RETURN: ret = Factorial of x

      FUNCTION LIB_FACUL(v_x)

      INTEGER*4 debug
      COMMON debug

      INTEGER*8 LIB_FACUL
      INTEGER*4 v_x, x

      INTEGER*8 i
      INTEGER*8 f
      x = v_x
      i = 1
      f = 1
      DO i = 1, x, 1
        f = (f * i)
        if (debug .eq. 1) THEN
          write(*,*) "FACUL:",f
        ENDIF
      END DO
      LIB_FACUL = f

      END
```

Listing A.5: lib.for

A.2.2 Source Code in Weak Unsafe Typed WSL

```
C:"Type-Level: WeakUnsafeTypedWSL";
VAR <
 INTEGER*4 :: debug := 0,
 INTEGER*4 :: MAX_IT := 100
>:
 VAR <
   REAL*8:: x := 1,
   INTEGER*4 :: n := 10
 >:
   ANALYSIS1( x,n );
   ANALYSIS2( n )
 ENDVAR
ENDVAR
```

Listing A.6: case_study.wsl

```
C:"Type-Level: WeakUnsafeTypedWSL";
BEGIN
 SKIP
WHERE
PROC ANALYSIS1( REAL*8 :: x,
 INTEGER*4 :: n) ==
 VAR <
   REAL*8 :: ret1 := f0
 >:
   PRINT("\nPartial Integration");
   PRINT("--------------------");
   BPARTINT1( x,n VAR ret1);
   PRINT("Backward:" ++ {STRING*0} ret1);
   FPARTINT1( x,n VAR ret1);
   PRINT("Forward :" ++ {STRING*0} ret1)
 ENDVAR
END
PROC ANALYSIS2( INTEGER*4 :: n ) ==
 VAR <
   INTEGER*4 :: f := 0,
   REAL*8 :: n1 := f0,
   REAL*8 :: n2 := f0,
   INTEGER*8 :: ret1 := 0,
   REAL*8 :: ret2 := f0,
   REAL*8 :: ret3 := f0,
   REAL*8 :: ret4 := f0,
   REAL*8 :: r_f := f0
```

```
 >:
   PRINT("\nANALYSIS2");
   PRINT("---------");
   FOR f := 0 TO n STEP 1 DO
      FACUL( f VAR ret1);
      PRINT("Facul:" ++ {STRING*0} ret1);
      n1 := {REAL*8} ret1;
      SQRT( n1 VAR ret2);
      PRINT("Sqrt :" ++ {STRING*0} ret2);
      r_f := {REAL*8} f;
      EXP( r_f VAR ret3);
      PRINT("Exp  :" ++ {STRING*0} ret3);
      n2 := ret3;
      SQRT( n2 VAR ret4);
      PRINT("Sqrt :" ++ {STRING*0} ret4)
   OD
 ENDVAR
END
END
```

Listing A.7: analysis.wsl

```
C:"Type-Level: WeakUnsafeTypedWSL";
BEGIN
 SKIP
WHERE
PROC BPARTINT1(   REAL*8 :: x,
 INTEGER*4 :: n VAR
 REAL*8 :: ret) ==
 C:"Procedure:BPARTINT1";
 C:"==================";
 C:"Partial Integration (backward) of t^n * e^-t to t";
 C:"PARAM: x = Upper limit of Integration (lower limit is 0)";
 C:"PARAM: n = power of t";
 C:"RETURN: ret = Result of integration";
 VAR <
   REAL*8 :: a := f0,
   REAL*8 :: expv := f0,
   REAL*8 :: xp := f0,
   INTEGER*4 :: i := 0,
   REAL*8 :: ret1 := f0
 >:
   EXP( x VAR ret1);
   expv := ret1;
   a := f1 - f1 / expv;
   FOR i := 1 TO n STEP 1 DO
```

```
      xp := x ** {REAL*8} i;
      a := a * {REAL*8} i - xp / expv;
      IF __GLOBAL__debug = 1 THEN
          PRINT("BPARTINT1:" ++ {STRING*0} a)
      FI
   OD;
   ret := a
 ENDVAR
END
PROC FPARTINT1(   REAL*8 :: x,
 INTEGER*4 :: n VAR
 REAL*8 :: ret) ==
 C:"Procedure:FPARTINT1";
 C:"==================";
 C:"Partial Integration (forward) of t^n * e^-t to t";
 C:"PARAM: x = Upper limit of Integration (lower limit is 0)";
 C:"PARAM: n = power of t";
 C:"RETURN: ret = Result of integration";
 VAR <
   REAL*8 :: a := f0,
   REAL*8 :: expv := f0,
   INTEGER*4 :: xp := f0,
   INTEGER*4 :: p := 0,
   INTEGER*4 :: i := 0,
   REAL*8 :: ret1 := f0
 >:
   EXP( x VAR ret1);
   expv := ret1;
   a := -x ** {REAL*8} n / expv;
   p := 1;
   FOR i := 0 TO n - 1 STEP 1 DO
      p := p * n - i;
      xp :=  (n - i - 1);
      a := a - {REAL*8} p * x ** {REAL*8} xp / expv;
      IF __GLOBAL__debug = 1 THEN
          PRINT("FPARTINT1:" ++ {STRING*0} a)
      FI
   OD;
   a := a + {REAL*8} p;
   ret := a
 ENDVAR
END
END
```

Listing A.8: part_int.wsl

```
C:"Type-Level: WeakUnsafeTypedWSL";
 BEGIN
   SKIP
 WHERE
 PROC SQRT( REAL*8 :: x VAR
   REAL*8 :: ret) ==
   C:"Procedure:SQRT";
   C:"==============";
   C:"A quick square root function";
   C:"PARAM: x = Number for square root";
   C:"RETURN: ret = Square root of x";
   VAR <
      REAL*8:: n := x / f2,
      REAL*8 :: xnew := f0,
      INTEGER*4 :: i := 0
   >:
      FOR i := 0 TO MAX_IT STEP 1 DO
          xnew := n + x / n / f2;
          IF xnew < x THEN
               n := xnew
          ELSIF TRUE THEN
               EXIT(1)
          FI;
          IF debug = 1 THEN
               PRINT("SQRT:" ++ {STRING*0} xnew)
          FI
      OD;
      ret := xnew
   ENDVAR
 END
 PROC EXP( REAL*8 :: x VAR
   REAL*8 :: ret) ==
   C:"Procedure:EXP";
   C:"=============";
   C:"Calculates powers of Euler's number";
   C:"PARAM: x = Power of e to calculate";
   C:"RETURN: ret = Euler's number to the power of x";
   VAR <
      INTEGER*4 :: i := 1,
      REAL*8 :: p := f1,
      REAL*8 :: s := f1,
      REAL*8 :: t := f2
   >:
      WHILE s <> t DO
          t := s;
```

```
          p := p * x / {REAL*8} i;
          s := s + p;
          i := i + 1;
          IF debug = 1 THEN
               PRINT("EXP:" ++ {STRING*0} s)
          FI
       OD;
       ret := s
    ENDVAR
 END
 PROC FACUL( INTEGER*4 :: x VAR
    INTEGER*8 :: ret) ==
    C:"Procedure:FACUL";
    C:"=============";
    C:"Calculates the factorial of x";
    C:"PARAM: x = Factorial to calculate";
    C:"RETURN: ret = Factorial of x";
    VAR <
       INTEGER*4 :: i := 1,
       INTEGER*8 :: f := 1
    >:
       FOR i := 1 TO x STEP 1 DO
          f := f * {INTEGER*8} i;
          IF debug = 1 THEN
               PRINT("FACUL:" ++ {STRING*0} f)
          FI
       OD;
       ret := f
    ENDVAR
 END
 END
ENDVAR
```

Listing A.9: lib.wsl

A.2.3 Source Code in Object Oriented Typed WSL

```
C:"Type-Level: OOTypedWSL";
BEGIN
  VAR <
   INTEGER*4 :: debug := 0,

   OSTRUCT :: ANALYSIS := <
             PUBLIC PROC ANALYSIS( )
   >,
   ANALYSIS :: OANALYSIS := < >,

   OSTRUCT :: ANALYSIS1 := <

       PUBLIC PROC ANALYSIS1( REAL*8, INTEGER*4 ),
       PUBLIC PROC BPARTINT1( REAL*8, INTEGER*4 VAR REAL*8 ),
       PUBLIC PROC FPARTINT1( REAL*8, INTEGER*4 VAR REAL*8 ),
   >,
   ANALYSIS1 :: OANALYSIS1 := < >,

   OSTRUCT :: ANALYSIS2 := <
       PUBLIC INTEGER*4 :: MAX_IT := 100,

       PUBLIC PROC ANALYSIS2( INTEGER*4 ),
       PUBLIC PROC FACUL( REAL*8 VAR REAL*8 ),
       PUBLIC PROC SQRT( REAL*8 VAR REAL*8 )
       PUBLIC PROC EXP( REAL*8 VAR REAL*8 )
   >,
   ANALYSIS2 :: OANALYSIS2 := < >
  >:

    !i OANALYSIS.ANALYSIS()

  ENDVAR
WHERE
PROC ANALYSIS( ) ==
 VAR <
   REAL*8:: x := 1,
   INTEGER*4 :: n := 10
 >:
   ANALYSIS1( x,n );
   ANALYSIS2( n )
 ENDVAR
END
```

```
END
```

Listing A.10: case_study.wsl

```
C:"Type-Level: OOTypedWSL";
BEGIN
 SKIP
WHERE
PROC ANALYSIS1( REAL*8 :: x,
 INTEGER*4 :: n) ==
 VAR <
   REAL*8 :: ret1 := f0
 >:
   PRINT("\nPartial Integration");
   PRINT("--------------------");
   !i ANALYSIS1.BPARTINT1( x,n VAR ret1);
   PRINT("Backward:" ++ {STRING*0} ret1);
   !i ANALYSIS1.FPARTINT1( x,n VAR ret1);
   PRINT("Forward :" ++ {STRING*0} ret1)
 ENDVAR
END
PROC ANALYSIS2( INTEGER*4 :: n) ==
 VAR <
   INTEGER*4 :: f := 0,
   REAL*8 :: n1 := f0,
   REAL*8 :: n2 := f0,
   INTEGER*8 :: ret1 := 0,
   REAL*8 :: ret2 := f0,
   REAL*8 :: ret3 := f0,
   REAL*8 :: ret4 := f0,
   REAL*8 :: r_f := f0
 >:
   PRINT("\nANALYSIS2");
   PRINT("---------");
   FOR f := 0 TO n STEP 1 DO
      !i ANALYSIS2.FACUL( f VAR ret1);
      PRINT("Facul:" ++ {STRING*0} ret1);
      n1 := {REAL*8} ret1;
      !i ANALYSIS2.SQRT( n1 VAR ret2);
      PRINT("Sqrt :" ++ {STRING*0} ret2);
      r_f := {REAL*8} f;
      !i ANALYSIS2.EXP( r_f VAR ret3);
      PRINT("Exp  :" ++ {STRING*0} ret3);
      n2 := ret3;
      !i ANALYSIS2.SQRT( n2 VAR ret4);
      PRINT("Sqrt :" ++ {STRING*0} ret4)
```

```
   OD
 ENDVAR
END
END
```

Listing A.11: analysis.wsl

```
C:"Type-Level: OOTypedWSL";
BEGIN
 SKIP
WHERE
PROC BPARTINT1( REAL*8 :: x,
 INTEGER*4 :: n VAR
 REAL*8 :: ret) ==
 C:"Procedure:BPARTINT1";
 C:"====================";
 C:"Partial Integration (backward) of t^n * e^-t to t";
 C:"PARAM: x = Upper limit of Integration (lower limit is 0)";
 C:"PARAM: n = power of t";
 C:"RETURN: ret = Result of integration";
 VAR <
   REAL*8 :: a := f0,
   REAL*8 :: expv := f0,
   REAL*8 :: xp := f0,
   INTEGER*4 :: i := 0,
   REAL*8 :: ret1 := f0
 >:
   !i ANALYSIS1.EXP( x VAR ret1);
   expv := ret1;
   a := f1 - f1 / expv;
   FOR i := 1 TO n STEP 1 DO
      xp := x ** {REAL*8} i;
      a := a * {REAL*8} i - xp / expv;
      IF debug = 1 THEN
          PRINT("BPARTINT1:" ++ {STRING*0} a)
      FI
   OD;
   ret := a
 ENDVAR
END
PROC FPARTINT1( REAL*8 :: x,
 INTEGER*4 :: n VAR
 REAL*8 :: ret) ==
 C:"Procedure:FPARTINT1";
 C:"====================";
 C:"Partial Integration (forward) of t^n * e^-t to t";
```

```
 C:"PARAM: x = Upper limit of Integration (lower limit is 0)";
 C:"PARAM: n = power of t";
 C:"RETURN: ret = Result of integration";
 VAR <
   REAL*8 :: a := f0,
   REAL*8 :: expv := f0,
   INTEGER*4 :: xp := f0,
   INTEGER*4 :: p := 0,
   INTEGER*4 :: i := 0,
   REAL*8 :: ret1 := f0
 >:
   !i ANALYSIS1.EXP( x VAR ret1);
   expv := ret1;
   a := -x ** {REAL*8} n / expv;
   p := 1;
   FOR i := 0 TO n - 1 STEP 1 DO
      p := p * n - i;
      xp :=  (n - i - 1);
      a := a - {REAL*8} p * x ** {REAL*8} xp / expv;
      IF debug = 1 THEN
          PRINT("FPARTINT1:" ++ {STRING*0} a)
      FI
   OD;
   a := a + {REAL*8} p;
   ret := a
 ENDVAR
END
END
```

Listing A.12: part_int.wsl

```
C:"Type-Level: OOTypedWSL";
BEGIN
   SKIP
WHERE
 PROC SQRT( REAL*8 :: x VAR
   REAL*8 :: ret) ==
   C:"Procedure:SQRT";
   C:"==============";
   C:"A quick square root function";
   C:"PARAM: x = Number for square root";
   C:"RETURN: ret = Square root of x";
   VAR <
      REAL*8:: n := x / f2,
      REAL*8 :: xnew := f0,
      INTEGER*4 :: i := 0
```

```
  >:
     FOR i := 0 TO MAX_IT STEP 1 DO
         xnew := n + x / n / f2;
         IF xnew < x THEN
              n := xnew
         ELSIF TRUE THEN
              EXIT(1)
         FI;
         IF debug = 1 THEN
              PRINT("SQRT:" ++ {STRING*0} xnew)
         FI
     OD;
     ret := xnew
  ENDVAR
END
PROC EXP( REAL*8 :: x VAR
  REAL*8 :: ret) ==
  C:"Procedure:EXP";
  C:"=============";
  C:"Calculates powers of Euler's number";
  C:"PARAM: x = Power of e to calculate";
  C:"RETURN: ret = Euler's number to the power of x";
  VAR <
     INTEGER*4 :: i := 1,
     REAL*8 :: p := f1,
     REAL*8 :: s := f1,
     REAL*8 :: t := f2
  >:
     WHILE s <> t DO
         t := s;
         p := p * x / {REAL*8} i;
         s := s + p;
         i := i + 1;
         IF debug = 1 THEN
              PRINT("EXP:" ++ {STRING*0} s)
         FI
     OD;
     ret := s
  ENDVAR
END
PROC FACUL( INTEGER*4 :: x VAR
  INTEGER*8 :: ret) ==
  C:"Procedure:FACUL";
  C:"=============";
  C:"Calculates the factorial of x";
```

```
   C:"PARAM: x = Factorial to calculate";
   C:"RETURN: ret = Factorial of x";
   VAR <
      INTEGER*4 :: i := 1,
      INTEGER*8 :: f := 1
   >:
      FOR i := 1 TO x STEP 1 DO
          f := f * {INTEGER*8} i;
          IF debug = 1 THEN
               PRINT("FACUL:" ++ {STRING*0} f)
          FI
      OD;
      ret := f
   ENDVAR
 END
END
```

Listing A.13: lib.wsl

A.2.4 Source Code in Java

```
import java.util.HashMap;

public class global {

        /**
         * Global Variables
         */
        public static int debug = 0;

        /**
         * Object registry of the system
         */
        private static HashMap<String, Object> obj = new HashMap<String, Object>();

        /**
         * Main Entry Point
         *
         * @param args
         *             Command line arguments
         */
        public static void main(String args[]) {
                // Instantiate Objects
                register("analysis", new analysis());
                register("analysis1", new analysis1());
                register("analysis2", new analysis2());

                // Start
                ((analysis) global.get("analysis")).ANALYSIS();
        }

        /**
         * Register an object
         *
         * @param name
         *             Name of object
         * @param o
         *             Object to register
         */
        public static void register(String name, Object o) {
                obj.put(name, o);
        }

        /**
```

```
         * Get a registered Object
         *
         * @param name
         *             Name of object
         * @return The requested object
         */
        public static Object get(String name) {
                return obj.get(name);
        }
}
```

Listing A.14: global.java

```
public class analysis {

        public void ANALYSIS() {
                double x = 1;
                int n = 10;
                ((analysis1) global.get("analysis1")).ANALYSIS1(x, n);
                ((analysis2) global.get("analysis2")).ANALYSIS2(n);
        }
}
```

Listing A.15: analysis.java

```
public class analysis1 {

        public void ANALYSIS1(double x, int n) {
                double ret1 = 0;
                System.out.println("\nPartial Integration");
                System.out.println("-------------------");
                ret1 = BPARTINT1(x, n);
                System.out.println("Backward:" + ret1);
                ret1 = FPARTINT1(x, n);
                System.out.println("Forward :" + ret1);
        }

        /**
         * Partial Integration (backward) of t^n * e^-t to t
         *
         * @param x
         *             Upper limit of Integration (lower limit is 0)
         * @param n
         *             Power of t
         * @return Result of integration
         */
```

```
public double BPARTINT1(double x, int n) {
        double a = 0;
        double expv = 0;
        double xp = 0;
        int i = 0;
        double ret1 = 0;
        ret1 = ((analysis2) global.get("analysis2")).EXP(x);
        expv = ret1;
        a = (1 - (1 / expv));
        for (i = 1; i < n + 1; i++) {
                xp = Math.pow(x, i);
                a = ((a * i) - (xp / expv));
                if (global.debug == 1) {
                        System.out.println("BPARTINT1:" + a);
                }
        }
        return a;
}

/**
 * Partial Integration (forward) of tˆn * eˆ-t to t
 *
 * @param x
 *             Upper limit of Integration (lower limit is 0)
 * @param n
 *             Power of t
 * @return Result of integration
 */
public double FPARTINT1(double x, int n) {
        double a = 0;
        double expv = 0;
        double xp = 0;
        int p = 0;
        int i = 0;
        double ret1 = 0;
        ret1 = ((analysis2) global.get("analysis2")).EXP(x);
        expv = ret1;
        a = (-Math.pow(x, n) / expv);
        p = 1;
        for (i = 0; i < n; i++) {
                p = (p * (n - i));
                xp = (n - i - 1);
                a = (a - (p * (Math.pow(x, xp) / expv)));
                if (global.debug == 1) {
                        System.out.println("FPARTINT1:" + a);
```

```
                        }
                }
                a = (a + p);
                return a;
        }
}
```

Listing A.16: analysis1.java

```
public class analysis2 {

        /**
         * Maximum Iterations
         */
        private int MAX_IT = 100;

        public void ANALYSIS2(int n) {
                int f = 0;
                double n1 = 0;
                double n2 = 0;
                long ret1 = 0;
                double ret2 = 0;
                double ret3 = 0;
                double ret4 = 0;
                System.out.println("\nANALYSIS2");
                System.out.println("---------");
                for (f = 0; f < n + 1; f++) {
                        ret1 = FACUL(f);
                        System.out.println("Facul:" + ret1);
                        n1 = ret1;
                        ret2 = SQRT(n1);
                        System.out.println("Sqrt :" + ret2);
                        ret3 = EXP(f);
                        System.out.println("Exp  :" + ret3);
                        n2 = ret3;
                        ret4 = SQRT(n2);
                        System.out.println("Sqrt :" + ret4);
                }
        }

        /**
         * Calculates the factorial of x
         *
         * @param x
         *            Factorial to calculate
         * @return Factorial of x
```

```
 */
public long FACUL(int x) {
        int i = 1;
        long f = 1;
        for (i = 1; i < x + 1; i++) {
                f = (f * i);
                if (global.debug == 1) {
                        System.out.println("FACUL:" + f);
                }
        }
        return f;
}

/**
 * A quick square root function";
 *
 * @param x
 *            Number for square root
 * @return Square root of x
 */
public double SQRT(double x) {
        double n = (x / 2);
        double xnew = 0;
        int i = 0;
        for (i = 0; i < MAX_IT + 1; i++) {
                xnew = ((n + (x / n)) / 2);
                if (xnew < x) {
                        n = xnew;
                } else {
                        break;
                }
                if (global.debug == 1) {
                        System.out.println("SQRT:" + xnew);
                }
        }
        return xnew;
}

/**
 * Calculates powers of Euler's number
 *
 * @param x
 *            Power of e to calculate
 * @return Euler's number to the power of x
 */
```

```
        public double EXP(double x) {
                int i = 1;
                double p = 1;
                double s = 1;
                double t = 2;
                while (s != t) {
                        t = s;
                        p = ((p * x) / i);
                        s = (s + p);
                        i = (i + 1);
                        if (global.debug == 1) {
                                System.out.println("EXP:" + s);
                        }
                }
                return s;
        }
}
```

Listing A.17: analysis.java

Appendix B

Data Structures for Code Analysis

B.1 Lexer Token List

LexicalToken	GroupName	Characters	ASCII-Code	ID
WHITESPACE	WhiteSpace		32	-1
WHITESPACE	WhiteSpace		9	-1
WHITESPACE	WhiteSpace		10	-1
SPECIALCHAR	SpecialChar		34	-1
SPECIALCHAR	SpecialChar		43	-1
SPECIALCHAR	SpecialChar		61	-1
SPECIALCHAR	SpecialChar		45	-1
SPECIALCHAR	SpecialChar		41	-1
SPECIALCHAR	SpecialChar		40	-1
SPECIALCHAR	SpecialChar		42	-1
SPECIALCHAR	SpecialChar		94	-1
SPECIALCHAR	SpecialChar		125	-1
SPECIALCHAR	SpecialChar		123	-1
SPECIALCHAR	SpecialChar		63	-1
SPECIALCHAR	SpecialChar		33	-1
SPECIALCHAR	SpecialChar		124	-1
SPECIALCHAR	SpecialChar		93	-1
SPECIALCHAR	SpecialChar		91	-1
SPECIALCHAR	SpecialChar		58	-1
SPECIALCHAR	SpecialChar		44	-1
SPECIALCHAR	SpecialChar		46	-1
SPECIALCHAR	SpecialChar		60	-1
SPECIALCHAR	SpecialChar		62	-1
SPECIALCHAR	SpecialChar		47	-1
SPECIALCHAR	SpecialChar		39	-1
SPECIALCHAR	SpecialChar		92	-1
SPECIALCHAR	SpecialChar		59	-1
S_ABORT	ReservedWord	ABORT		997
S_ABS	ReservedWord	ABS		100
S_ACTION	ReservedWord	Action		536
S_ACTIONS	ReservedWord	ACTIONS		69
S_AND	ReservedWord	AND		31
S_ARRAY	ReservedWord	ARRAY		94
S_ASSIGN	ReservedWord	Assign		533
S_ASSIGNS	ReservedWord	Assigns		534
S_ATEACH	ReservedWord	ATEACH		511
S_BEGIN	ReservedWord	BEGIN		500
S_BFUNCT	ReservedWord	BFUNCT		504
S_BUTLAST	ReservedWord	BUTLAST		111
S_CALL	ReservedWord	CALL		68

```
S_COMMENT       ReservedWord   Comment                34
S_COMMENT       ReservedWord   COMMENT                34
S_COND_PLACE    ReservedWord   $Condition$            803
S_CONDITION     ReservedWord   Condition              524
S_D_DO          ReservedWord   D_DO                   47
S_D_IF          ReservedWord   D_IF                   45
S_DEFINITION    ReservedWord   Definition             527
S_DEFINITIONS   ReservedWord   Definitions            528
S_DIV           ReservedWord   DIV                    63
S_DO            ReservedWord   DO                     28
S_ELSE          ReservedWord   ELSE                   25
S_ELSIF         ReservedWord   ELSIF                  24
S_EMPTY         ReservedWord   EMPTY                  59
S_END           ReservedWord   END                    75
S_ENDACTIONS    ReservedWord   ENDACTIONS             70
S_ENDFILL       ReservedWord   ENDFILL                116
S_ENDJOIN       ReservedWord   ENDJOIN                86
S_ENDMATCH      ReservedWord   ENDMATCH               513
S_ENDSPEC       ReservedWord   ENDSPEC                509
S_ENDVAR        ReservedWord   ENDVAR                 64
S_ERROR         ReservedWord   ERROR                  122
S_EVEN          ReservedWord   EVEN                   57
S_EXISTS        ReservedWord   EXISTS                 61
S_EXIT          ReservedWord   EXIT                   65
S_EXPN_PLACE    ReservedWord   $Expn$                 801
S_EXPRESSION    ReservedWord   Expression             522
S_EXPRESSIONS   ReservedWord   Expressions            523
S_FALSE         ReservedWord   FALSE                  50
S_FI            ReservedWord   FI                     26
S_FILL          ReservedWord   FILL                   115
S_FILL2         ReservedWord   FILL2                  123
S_FOR           ReservedWord   FOR                    71
S_FORALL        ReservedWord   FORALL                 60
S_FOREACH       ReservedWord   FOREACH                510
S_FRAC          ReservedWord   FRAC                   102
S_FUNCT         ReservedWord   FUNCT                  503
S_GLOBAL        ReservedWord   Global                 538
S_GUARDED       ReservedWord   Guarded                535
S_HASH_TABLE    ReservedWord   HASH_TABLE             804
S_HEAD          ReservedWord   HEAD                   108
S_IF            ReservedWord   IF                     22
S_IFMATCH       ReservedWord   IFMATCH                512
S_IFMATCH2      ReservedWord   IFMATCH2               514
S_IN            ReservedWord   IN                     53
```

```
S_INDEX         ReservedWord   INDEX                120
S_INT           ReservedWord   INT                  101
S_INTS          ReservedWord   %Z                   91
S_JOIN          ReservedWord   JOIN                 85
S_LAST          ReservedWord   LAST                 110
S_LENGTH        ReservedWord   LENGTH               114
S_LVALUE        ReservedWord   Lvalue               529
S_LVALUES       ReservedWord   Lvalues              530
S_MAP           ReservedWord   MAP                  106
S_MAPHASH       ReservedWord   MAPHASH              121
S_MAX           ReservedWord   MAX                  105
S_MEMBER        ReservedWord   MEMBER               55
S_MIN           ReservedWord   MIN                  104
S_MOD           ReservedWord   MOD                  62
S_MW_BFUNCT     ReservedWord   MW_BFUNCT            507
S_MW_FUNCT      ReservedWord   MW_FUNCT             506
S_MW_PROC       ReservedWord   MW_PROC              505
S_NAS           ReservedWord   NAS                  532
S_NATS          ReservedWord   %N                   90
S_NOT           ReservedWord   NOT                  33
S_NOTIN         ReservedWord   NOTIN                54
S_NUMBERQ       ReservedWord   NUMBER               97
S_OD            ReservedWord   OD                   29
S_ODD           ReservedWord   ODD                  58
S_OR            ReservedWord   OR                   32
S_POP           ReservedWord   POP                  84
S_POWERSET      ReservedWord   POWERSET             113
S_PRINFLUSH     ReservedWord   PRINFLUSH            89
S_PRINT         ReservedWord   PRINT                88
S_PROC          ReservedWord   PROC                 502
S_PUSH          ReservedWord   PUSH                 83
S_RATS          ReservedWord   %Q                   92
S_REALS         ReservedWord   %R                   93
S_REDUCE        ReservedWord   REDUCE               107
S_RETURNS       ReservedWord   RETURNS              81
S_REVERSE       ReservedWord   REVERSE              112
S_SEQUENCE      ReservedWord   SEQUENCE             95
S_SGN           ReservedWord   SGN                  103
S_SKIP          ReservedWord   SKIP                 48
S_SLENGTH       ReservedWord   SLENGTH              118
S_SPEC          ReservedWord   SPEC                 508
S_STAT_PLACE    ReservedWord   $Statement$          800
S_STATEMENT     ReservedWord   Statement            520
S_STATEMENTS    ReservedWord   Statements           521
```

S_STEP	ReservedWord	STEP		73
S_STRINGBF	ReservedWord	STRING		96
S_STS	ReservedWord	STS		531
S_SUBSET	ReservedWord	SUBSET		56
S_SUBSTR	ReservedWord	SUBSTR		119
S_TAIL	ReservedWord	TAIL		109
S_TERMINAL	ReservedWord	Terminal		537
S_THEN	ReservedWord	THEN		23
S_TO	ReservedWord	TO		72
S_TRUE	ReservedWord	TRUE		49
S_VAR	ReservedWord	VAR		30
S_VAR_PLACE	ReservedWord	Var		802
S_VARIABLE	ReservedWord	Variable		526
S_WHERE	ReservedWord	WHERE		501
S_WHILE	ReservedWord	WHILE		27
S_ARROW	SpecialToken		45;62	41
S_AT	SpecialToken		64	74
S_AT_PAT_ONE	SpecialToken		64;126;63	703
S_BACKSLASH	SpecialToken		92	44
S_BECOMES	SpecialToken		58;61	1
S_BOX	SpecialToken		91;93	40
S_CARET	SpecialToken		94	117
S_COLON	SpecialToken		58	8
S_COMMA	SpecialToken		44	12
S_CONCAT	SpecialToken		43;43	20
S_DEFINE	SpecialToken		61;61	21
S_DOTDOT	SpecialToken		46;46	66
S_DOTSPACE	SpecialToken		46;32	87
S_EOF	SpecialToken			999
S_EQUAL	SpecialToken		61	2
S_EXPONENT	SpecialToken		42;42	46
S_FULLSTOP	SpecialToken		46	52
S_GEQ	SpecialToken		62;61	39
S_IDENTIFIER	SpecialToken			35
S_INTERSECT	SpecialToken		47;92	42
S_INVALID	SpecialToken			998
S_LANGLE	SpecialToken		60	15
S_LBRACE	SpecialToken		123	17
S_LBRACKET	SpecialToken		91	13
S_LEQ	SpecialToken		60;61	38
S_LPAREN	SpecialToken		40	10
S_MINUS	SpecialToken		45	4
S_NEQ	SpecialToken		60;62	9
S_NUMBER	SpecialToken			36

S_PAT_ANY	SpecialToken	126;42	702
S_PAT_MANY	SpecialToken	126;43	701
S_PAT_ONE	SpecialToken	126;63	700
S_PLINK_P	SpecialToken	33;80	77
S_PLINK_XC	SpecialToken	33;88;67	80
S_PLINK_XF	SpecialToken	33;88;70	79
S_PLINK_XP	SpecialToken	33;88;80	78
S_PLUS	SpecialToken	43	3
S_PRIME	SpecialToken	39	805
S_QUERY	SpecialToken	63	51
S_QUOTES	SpecialToken	34	19
S_RANGLE	SpecialToken	62	16
S_RBRACE	SpecialToken	125	18
S_RBRACKET	SpecialToken	93	14
S_RPAREN	SpecialToken	41	11
S_SCOPAREN	SpecialToken		76
S_SEMICOLON	SpecialToken	59	7
S_SLASH	SpecialToken	47	6
S_STRING	SpecialToken		37
S_TIMES	SpecialToken	42	5
S_UNION	SpecialToken	92;47	43
S_VBAR	SpecialToken	124	82

B.2 Lexer Table

No	CLevel	Ident	Char	ASCII	RegExp	TGroup	Token	Value
0	0					WhiteSpace		
1	0		?				S_QUERY	
2	0		,				S_COMMA	
3	0		]				S_RBRACKET	
4	0		{				S_LBRACE	
5	0		}				S_RBRACE	
6	0		(				S_LPAREN	
7	0		)				S_RPAREN	
8	0		|				S_VBAR	
9	0		;				S_SEMICOLON	
10	0		^				S_CARET	
11	0		=					
11	0.1		=				S_DEFINE	
11	0.2	1					S_EQUAL	
12	0		:					
12	0.1		=				S_BECOMES	
12	0.2	1					S_COLON	
13	0		.					
13	0.1		.				S_DOTDOT	
13	0.2					WhiteSpace	S_DOTSPACE	
13	0.3	1					S_FULLSTOP	
14	0		+					
14	0.1		+				S_CONCAT	
14	0.2	1					S_PLUS	
15	0		-					
15	0.1			62			S_ARROW	
15	0.2	1					S_MINUS	
16	0		*					
16	0.1		*				S_EXPONENT	
16	0.2	1					S_TIMES	
17	0		[					
17	0.1		]				S_BOX	
17	0.2	1					S_LBRACKET	
18	0			60				
18	0.1			62			S_NEQ	
18	0.2		=				S_LEQ	
18	0.3	1					S_LANGLE	
19	0			62				
19	0.1		=				S_GEQ	
19	0.2	1					S_RANGLE	

```
20  0              !
20  0.1            P                                  S_PLINK_P
20  0.2            X
20  0.2.1          P                                  S_PLINK_XP
20  0.2.2          F                                  S_PLINK_XF
20  0.2.3          C                                  S_PLINK_XC
20  0.2.4   1                                         S_INVALID
20  0.3     1                                         S_INVALID
21  0              ~
21  0.1            ?                                  S_PAT_ONE
21  0.2            +                                  S_PAT_MANY
21  0.3            *                                  S_PAT_ANY
21  0.4     1                                         S_INVALID
22  0              @
22  0.1            ~
22  0.1.1          ?                                  S_AT_PAT_ONE   SCAN_IDENTIFIER
22  0.1.2   1                                         S_INVALID      SCAN_IDENTIFIER
22  0.2     1                                         S_AT           SCAN_IDENTIFIER
23  0              C
23  0.1            :
23  0.1.1                  34                         S_COMMENT      SCAN_COMMENT
24  0              /
24  0.1            \                                  S_INTERSECT
24  0.2     1                                         S_SLASH
25  0              \
25  0.1            /                                  S_UNION
25  0.2     1                                         S_BACKSLASH
26  0                      34                         S_STRING       SCAN_STRING
27  0                              [0-9]              S_NUMBER       SCAN_NUMBER
28  0              '                                  S_PRIME
29  0                                      ReservedWord               SCAN_RESERVED_WORD
30  0       1                                         S_IDENTIFIER   SCAN_IDENTIFIER
```

B.3 Tree Node List

ID	Name	Syntax Name	General Type	Allowed children	Has value
General_Types					
1	T_Statement	Statement	0		0
2	T_Expression	Expression	0		0
3	T_Condition	Condition	0		0
4	T_Definition	Definition	0		0
5	T_Lvalue	Lvalue	0		0
6	T_Assign	Assign	0	5;2	0
7	T_Guarded	Guarded	0	3;17	0
8	T_Action	Action	0	9;17	0
9	T_Name	Name	0		1
Group_Types					
10	T_Expressions	Expressions	0	2	0
12	T_Lvalues	Lvalues	0	5	0
13	T_Assigns	Assigns	0	6	0
14	T_Definitions	Definitions	0	4	0
15	T_Actions	Actions	0	8	0
16	T_Guardeds	Guardeds	0		0
17	T_Statements	Statements	0	1	0
Specific_Types					
101	T_A_Proc_Call	A_Proc_Call	1	9;10;12	0
102	T_MW_Proc_Call	MW_Proc_Call	1	9;10;12	0
103	T_X_Proc_Call	X_Proc_Call	1	9;10	0
104	T_Stat_Place	Stat_Place	1		0
105	T_Stat_Pat_One	Stat_Pat_One	1		1
106	T_Stat_Pat_Many	Stat_Pat_Many	1		1
107	T_Stat_Pat_Any	Stat_Pat_Any	1		1
108	T_Abort	Abort	1		0
109	T_Assert	Assert	1	3	0
110	T_Assignment	Assignment	1	6	0
111	T_A_S	A_S	1	9;15	0
112	T_Call	Call	1		1
113	T_Comment	Comment	1		1
114	T_Cond	Cond	1	7	0
115	T_D_If	D_If	1	7	0
116	T_D_Do	D_Do	1	7	0
117	T_Exit	Exit	1		1
118	T_For	For	1	5;2;2;2;17	0

```
119  T_Foreach_Stat         Foreach_Stat         1        17             0
120  T_Foreach_Stats        Foreach_Stats        1        17             0
121  T_Foreach_TS           Foreach_TS           1        17             0
122  T_Foreach_TSs          Foreach_TSs          1        17             0
123  T_Foreach_STS          Foreach_STS          1        17             0
124  T_Foreach_Expn         Foreach_Expn         1        17             0
125  T_Foreach_Cond         Foreach_Cond         1        17             0
126  T_Ateach_Stat          Ateach_Stat          1        17             0
127  T_Ateach_Stats         Ateach_Stats         1        17             0
128  T_Ateach_TS            Ateach_TS            1        17             0
129  T_Ateach_TSs           Ateach_TSs           1        17             0
130  T_Ateach_STS           Ateach_STS           1        17             0
131  T_Ateach_Expn          Ateach_Expn          1        17             0
132  T_Ateach_Cond          Ateach_Cond          1        17             0
133  T_Floop                Floop                1        17             0
134  T_Join                 Join                 1        17;17          0
135  T_Pop                  Pop                  1        5;5            0
136  T_Proc_Call            Proc_Call            1        9;10;12        0
137  T_Push                 Push                 1        5;2            0
138  T_Spec                 Spec                 1        12;3           0
139  T_Var                  Var                  1        13;17          0
140  T_Where                Where                1        17;14          0
141  T_While                While                1        3;17           0
142  T_MW_Proc              MW_Proc              1        9;12;12;17     0
143  T_MW_Funct             MW_Funct             1        9;12;13;17;2   0
144  T_MW_BFunct            MW_BFunct            1        9;12;13;17;3   0
145  T_Skip                 Skip                 1                       0
146  T_Foreach_NAS          Foreach_NAS          1        17             0
147  T_Ateach_NAS           Ateach_NAS           1        17             0
148  T_Foreach_Variable     Foreach_Variable     1        17             0
149  T_Foreach_Global_Var   Foreach_Global_Var   1        17             0
150  T_Ateach_Variable      Ateach_Variable      1        17             0
151  T_Ateach_Global_Var    Ateach_Global_Var    1        17             0
152  T_Foreach_Lvalue       Foreach_Lvalue       1        17             0
153  T_Ateach_Lvalue        Ateach_Lvalue        1        17             0
154  T_For_In               For_In               1        5;2;17         0
155  T_Puthash              Puthash              1        5;2;2          0
156  T_Print                Print                1        10             0
157  T_Prinflush            Prinflush            1        10             0
158  T_Maphash              Maphash              1        9;2            0
159  T_Error                Error                1        10             0
160  T_Stat_Int_One         Stat_Int_One         1        2              0
161  T_Stat_Int_Any         Stat_Int_Any         1        2              0
162  T_Stat_Val_One         Stat_Val_One         1                       0
```

```
163  T_Stat_Val_Any        Stat_Val_Any        1          0
166  T_Ifmatch2_Stat       Ifmatch2_Stat       1  1;17;17 0
167  T_Ifmatch2_Expn       Ifmatch2_Expn       1  2;17;17 0
168  T_Ifmatch2_Cond       Ifmatch2_Cond       1  3;17;17 0
169  T_Ifmatch2_Defn       Ifmatch2_Defn       1  4;17;17 0
170  T_Ifmatch2_Lvalue     Ifmatch2_Lvalue     1  5;17;17 0
171  T_Ifmatch2_Assign     Ifmatch2_Assign     1  6;17;17 0
172  T_Ifmatch2_Guarded    Ifmatch2_Guarded    1  7;17;17 0
173  T_Ifmatch2_Action     Ifmatch2_Action     1  8;17;17 0
174  T_Ifmatch2_Stats      Ifmatch2_Stats      1  17;17;17 0
175  T_Ifmatch2_Expns      Ifmatch2_Expns      1  10;17;17 0
177  T_Ifmatch2_Lvalues    Ifmatch2_Lvalues    1  12;17;17 0
178  T_Ifmatch2_Assigns    Ifmatch2_Assigns    1  13;17;17 0
179  T_Ifmatch2_Defns      Ifmatch2_Defns      1  14;17;17 0
180  T_Ifmatch_Stat        Ifmatch_Stat        1  1;17;17 0
181  T_Ifmatch_Expn        Ifmatch_Expn        1  2;17;17 0
182  T_Ifmatch_Cond        Ifmatch_Cond        1  3;17;17 0
183  T_Ifmatch_Defn        Ifmatch_Defn        1  4;17;17 0
184  T_Ifmatch_Lvalue      Ifmatch_Lvalue      1  5;17;17 0
185  T_Ifmatch_Assign      Ifmatch_Assign      1  6;17;17 0
186  T_Ifmatch_Guarded     Ifmatch_Guarded     1  7;17;17 0
187  T_Ifmatch_Action      Ifmatch_Action      1  8;17;17 0
188  T_Ifmatch_Stats       Ifmatch_Stats       1  17;17;17 0
189  T_Ifmatch_Expns       Ifmatch_Expns       1  10;17;17 0
191  T_Ifmatch_Lvalues     Ifmatch_Lvalues     1  12;17;17 0
192  T_Ifmatch_Assigns     Ifmatch_Assigns     1  13;17;17 0
193  T_Ifmatch_Defns       Ifmatch_Defns       1  14;17;17 0
201  T_X_Funct_Call        X_Funct_Call        2  9;10    0
202  T_MW_Funct_Call       MW_Funct_Call       2  9;10    0
203  T_Expn_Place          Expn_Place          2          0
204  T_Var_Place           Var_Place           2          0
205  T_Number              Number              2          1
206  T_String              String              2          1
207  T_Variable            Variable            2          1
208  T_Primed_Var          Primed_Var          2          1
209  T_Sequence            Sequence            2  10      0
210  T_Aref                Aref                2  2;10    0
211  T_Sub_Seg             Sub_Seg             2  2;2;2   0
212  T_Rel_Seg             Rel_Seg             2  2;2;2   0
213  T_Final_Seg           Final_Seg           2  2;2     0
214  T_Funct_Call          Funct_Call          2  9;10    0
215  T_Map                 Map                 2  9;2     0
216  T_Reduce              Reduce              2  9;2     0
217  T_Expn_Pat_One        Expn_Pat_One        2          1
```

```
218  T_Expn_Pat_Many      Expn_Pat_Many      2                  1
219  T_Expn_Pat_Any       Expn_Pat_Any       2                  1
220  T_Plus               Plus               2        2         0
221  T_Minus              Minus              2        2         0
222  T_Times              Times              2        2         0
223  T_Divide             Divide             2        2         0
224  T_Exponent           Exponent           2        2         0
225  T_Mod                Mod                2        2;2       0
226  T_Div                Div                2        2;2       0
227  T_If                 If                 2        3;2;2     0
228  T_Abs                Abs                2        2         0
229  T_Frac               Frac               2        2         0
230  T_Int                Int                2        2         0
231  T_Sgn                Sgn                2        2         0
232  T_Max                Max                2        2         0
233  T_Min                Min                2        2         0
234  T_Intersection       Intersection       2        2         0
235  T_Union              Union              2        2         0
236  T_Set_Diff           Set_Diff           2        2;2       0
237  T_Powerset           Powerset           2        2         0
238  T_Set                Set                2        2;3       0
239  T_Array              Array              2        2;2       0
240  T_Head               Head               2        2         0
241  T_Tail               Tail               2        2         0
242  T_Last               Last               2        2         0
243  T_Butlast            Butlast            2        2         0
244  T_Length             Length             2        2         0
245  T_Reverse            Reverse            2        2         0
246  T_Concat             Concat             2        2         0
251  T_Negate             Negate             2        2         0
252  T_Invert             Invert             2        2         0
253  T_Struct             Struct             2        9;2       0
254  T_Get_n              Get_n              2        2;2       0
255  T_Get                Get                2        2;2       0
256  T_Gethash            Gethash            2        2;2       0
257  T_Hash_Table         Hash_Table         2                  0
258  T_Slength            Slength            2        2         0
259  T_Substr             Substr             2        10        0
260  T_Index              Index              2        10        0
261  T_Expn_Int_One       Expn_Int_One       2        2         0
262  T_Expn_Int_Any       Expn_Int_Any       2        2         0
263  T_Expn_Val_One       Expn_Val_One       2                  0
264  T_Expn_Val_Any       Expn_Val_Any       2                  0
265  T_Fill2_Stat         Fill2_Stat         2        1         0
```

```
266  T_Fill2_Expn          Fill2_Expn          2       2          0
267  T_Fill2_Cond          Fill2_Cond          2       3          0
268  T_Fill2_Defn          Fill2_Defn          2       4          0
269  T_Fill2_Lvalue        Fill2_Lvalue        2       5          0
270  T_Fill2_Assign        Fill2_Assign        2       6          0
271  T_Fill2_Guarded       Fill2_Guarded       2       7          0
272  T_Fill2_Action        Fill2_Action        2       8          0
273  T_Fill2_Stats         Fill2_Stats         2       17         0
274  T_Fill2_Expns         Fill2_Expns         2       10         0
276  T_Fill2_Lvalues       Fill2_Lvalues       2       12         0
277  T_Fill2_Assigns       Fill2_Assigns       2       13         0
278  T_Fill2_Defns         Fill2_Defns         2       14         0
281  T_Fill_Stat           Fill_Stat           2       1          0
282  T_Fill_Expn           Fill_Expn           2       2          0
283  T_Fill_Cond           Fill_Cond           2       3          0
284  T_Fill_Defn           Fill_Defn           2       4          0
285  T_Fill_Lvalue         Fill_Lvalue         2       5          0
286  T_Fill_Assign         Fill_Assign         2       6          0
287  T_Fill_Guarded        Fill_Guarded        2       7          0
288  T_Fill_Action         Fill_Action         2       8          0
289  T_Fill_Stats          Fill_Stats          2       17         0
290  T_Fill_Expns          Fill_Expns          2       10         0
292  T_Fill_Lvalues        Fill_Lvalues        2       12         0
293  T_Fill_Assigns        Fill_Assigns        2       13         0
294  T_Fill_Defns          Fill_Defns          2       14         0
301  T_X_BFunct_Call       X_BFunct_Call       3       9;10       0
302  T_MW_BFunct_Call      MW_BFunct_Call      3       9;10       0
303  T_Cond_Place          Cond_Place          3                  0
304  T_BFunct_Call         BFunct_Call         3       9;10       0
305  T_Cond_Pat_One        Cond_Pat_One        3                  1
306  T_Cond_Pat_Many       Cond_Pat_Many       3                  1
307  T_Cond_Pat_Any        Cond_Pat_Any        3                  1
308  T_True                True                3                  0
309  T_False               False               3                  0
310  T_And                 And                 3       3          0
311  T_Or                  Or                  3       3          0
312  T_Not                 Not                 3       3          0
313  T_Equal               Equal               3       2;2        0
314  T_Less                Less                3       2;2        0
315  T_Greater             Greater             3       2;2        0
316  T_Less_Eq             Less_Eq             3       2;2        0
317  T_Greater_Eq          Greater_Eq          3       2;2        0
318  T_Not_Equal           Not_Equal           3       2;2        0
319  T_Even                Even                3       2          0
```

```
320  T_Odd                Odd                3        2              0
321  T_Empty              Empty              3        2              0
322  T_Subset             Subset             3        2;2            0
323  T_Member             Member             3        2;2            0
324  T_Forall             Forall             3        12;3           0
325  T_Exists             Exists             3        12;3           0
326  T_Implies            Implies            3        3;3            0
327  T_Sequenceq          Sequenceq          3        2              0
328  T_Numberq            Numberq            3        2              0
329  T_Stringq            Stringq            3        2              0
330  T_In                 In                 3        2;2            0
331  T_Not_In             Not_In             3        2;2            0
332  T_Cond_Int_One       Cond_Int_One       3        2              0
333  T_Cond_Int_Any       Cond_Int_Any       3        2              0
334  T_Cond_Val_One       Cond_Val_One       3                       0
335  T_Cond_Val_Any       Cond_Val_Any       3                       0
401  T_Proc               Proc               4        9;12;12;17     0
402  T_Funct              Funct              4        9;12;13;2      0
403  T_BFunct             BFunct             4        9;12;13;3      0
404  T_Defn_Pat_One       Defn_Pat_One       4                       1
405  T_Defn_Pat_Many      Defn_Pat_Many      4                       1
406  T_Defn_Pat_Any       Defn_Pat_Any       4                       1
407  T_Defn_Int_One       Defn_Int_One       4        2              0
408  T_Defn_Int_Any       Defn_Int_Any       4        2              0
409  T_Defn_Val_One       Defn_Val_One       4                       0
410  T_Defn_Val_Any       Defn_Val_Any       4                       0
501  T_Var_Lvalue         Var_Lvalue         5                       1
502  T_Aref_Lvalue        Aref_Lvalue        5        5;10           0
503  T_Sub_Seg_Lvalue     Sub_Seg_Lvalue     5        5;2;2          0
504  T_Rel_Seg_Lvalue     Rel_Seg_Lvalue     5        5;2;2          0
505  T_Final_Seg_Lvalue   Final_Seg_Lvalue   5        5;2            0
506  T_Lvalue_Pat_One     Lvalue_Pat_One     5                       1
507  T_Lvalue_Pat_Many    Lvalue_Pat_Many    5                       1
508  T_Lvalue_Pat_Any     Lvalue_Pat_Any     5                       1
509  T_Struct_Lvalue      Struct_Lvalue      5        9;5            0
510  T_Lvalue_Int_One     Lvalue_Int_One     5        2              0
511  T_Lvalue_Int_Any     Lvalue_Int_Any     5        2              0
512  T_Lvalue_Val_One     Lvalue_Val_One     5                       0
513  T_Lvalue_Val_Any     Lvalue_Val_Any     5                       0
601  T_Assign_Pat_One     Assign_Pat_One     6                       1
602  T_Assign_Pat_Any     Assign_Pat_Any     6                       1
603  T_Assign_Pat_Many    Assign_Pat_Many    6                       1
604  T_Assign_Int_One     Assign_Int_One     6        2              0
605  T_Assign_Int_Any     Assign_Int_Any     6        2              0
```

```
606   T_Assign_Val_One      Assign_Val_One      6              0
607   T_Assign_Val_Any      Assign_Val_Any      6              0
701   T_Guarded_Pat_One     Guarded_Pat_One     7              1
702   T_Guarded_Pat_Any     Guarded_Pat_Any     7              1
703   T_Guarded_Pat_Many    Guarded_Pat_Many    7              1
704   T_Guarded_Int_One     Guarded_Int_One     7        2     0
705   T_Guarded_Int_Any     Guarded_Int_Any     7        2     0
706   T_Guarded_Val_One     Guarded_Val_One     7              0
707   T_Guarded_Val_Any     Guarded_Val_Any     7              0
801   T_Action_Pat_One      Action_Pat_One      8              1
802   T_Action_Pat_Any      Action_Pat_Any      8              1
803   T_Action_Pat_Many     Action_Pat_Many     8              1
804   T_Action_Int_One      Action_Int_One      8        2     0
805   T_Action_Int_Any      Action_Int_Any      8        2     0
806   T_Action_Val_One      Action_Val_One      8              0
807   T_Action_Val_Any      Action_Val_Any      8              0
901   T_Name_Pat_One        Name_Pat_One        9              1
902   T_Name_Int_One        Name_Int_One        9        2     0
903   T_Name_Val_One        Name_Val_One        9              0
```

B.4 Tree Node List Continuation (PrettyPrintTemplate Column)

Name	PrettyPrintTemplate
General_Types	
T_Statement	
T_Expression	
T_Condition	
T_Definition	
T_Lvalue	
T_Assign	(C0); ;S_BECOMES; ;(C1)
T_Guarded	(C0); ;PARENT(T_Cond:S_THEN);PARENT(T_Ifmatch_Guarded:S_THEN) ;PARENT(T_Ifmatch2_Guarded:S_THEN);PARENT(T_D_If:S_ARROW); PARENT(T_D_Do:S_ARROW);\n;(C1)
T_Action	(C0); ;S_DEFINE;\n;(C1);S_END;\n
T_Name	(V)
Group_Types	
T_Expressions	(A,S_COMMA)
T_Lvalues	(A,S_COMMA)
T_Assigns	;(A,S_COMMA#);
T_Definitions	(A,\n);\n
T_Actions	(A,)
T_Guardeds	(A,)
T_Statements	I;(A,S_SEMICOLON#\n);\n
Specific_Types	
T_A_Proc_Call	S_PLINK_P; ;(C0);S_LPAREN;(C1); ;S_VAR; ;(C2);S_RPAREN
T_MW_Proc_Call	(C0);S_LPAREN;(C1);S_RPAREN
T_X_Proc_Call	S_PLINK_XP; ;(C0);S_LPAREN;(C1);S_RPAREN
T_Stat_Place	S_STAT_PLACE
T_Stat_Pat_One	S_PAT_ONE;(V)
T_Stat_Pat_Many	S_PAT_MANY;(V)
T_Stat_Pat_Any	S_PAT_ANY;(V)
T_Abort	S_ABORT
T_Assert	S_LBRACE;(C0);S_RBRACE
T_Assignment	CS:S_LANGLE;(A,S_COMMA#);CS:S_RANGLE
T_A_S	S_ACTIONS; ;(C0);S_COLON;\n;(A[1],);S_ENDACTIONS
T_Call	S_CALL; ;(V)
T_Comment	C:67;S_COLON;C:34;(V);C:34
T_Cond	S_IF; ;(C0);S_ELSIF; ;(A[1],S_ELSIF#);S_FI
T_D_If	S_D_IF; ;(C0);S_BOX; ;(A[1],S_BOX#);S_FI
T_D_Do	S_D_DO; ;(C0);S_BOX; ;(A[1],S_BOX#);S_OD
T_Exit	S_EXIT;S_LPAREN;(V);S_RPAREN
T_For	S_FOR; ;(C0); ;S_BECOMES; ;(C1); ;S_TO; ;(C2); ;S_STEP; ; (C3); ;S_DO;\n;(C4);S_OD

```
T_Foreach_Stat          S_FOREACH; ;S_STATEMENT; ;S_DO;\n;(A,);S_OD
T_Foreach_Stats         S_FOREACH; ;S_STATEMENTS; ;S_DO;\n;(A,);S_OD
T_Foreach_TS            S_FOREACH; ;S_TERMINAL; ;S_STATEMENT; ;S_DO;\n;(A,);S_OD
T_Foreach_TSs           S_FOREACH; ;S_TERMINAL; ;S_STATEMENTS; ;S_DO;\n;(A,);S_OD
T_Foreach_STS           S_FOREACH; ;S_STS; ;S_DO;\n;(A,);S_OD
T_Foreach_Expn          S_FOREACH; ;S_EXPRESSION; ;S_DO;\n;(A,);S_OD
T_Foreach_Cond          S_FOREACH; ;S_CONDITION; ;S_DO;\n;(A,);S_OD
T_Ateach_Stat           S_ATEACH; ;S_STATEMENT; ;S_DO;\n;(A,);S_OD
T_Ateach_Stats          S_ATEACH; ;S_STATEMENTS; ;S_DO;\n;(A,);S_OD
T_Ateach_TS             S_ATEACH; ;S_TERMINAL; ;S_STATEMENT; ;S_DO;\n;(A,);S_OD
T_Ateach_TSs            S_ATEACH; ;S_TERMINAL; ;S_STATEMENTS; ;S_DO;\n;(A,);S_OD
T_Ateach_STS            S_ATEACH; ;S_STS; ;S_DO;\n;(A,);S_OD
T_Ateach_Expn           S_ATEACH; ;S_EXPRESSION; ;S_DO;\n;(A,);S_OD
T_Ateach_Cond           S_ATEACH; ;S_CONDITION; ;S_DO;\n;(A,);S_OD
T_Floop                 S_DO;\n;(C0);S_OD
T_Join                  S_JOIN;\n;(C0);S_COMMA;\n;(C1);S_ENDJOIN
T_Pop                   S_POP;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Proc_Call             (C0);S_LPAREN;(C1); ;S_VAR; ;(C2);S_RPAREN
T_Push                  S_PUSH;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Spec                  S_SPEC; ;S_LANGLE;(C0);S_RANGLE;S_COLON; ;(C1); ;S_ENDSPEC
T_Var                   S_VAR; ;S_LANGLE;(C0);S_RANGLE;S_COLON;\n;(C1);S_ENDVAR
T_Where                 S_BEGIN;\n;(C0);S_WHERE;\n;(C1);S_END
T_While                 S_WHILE; ;(C0); ;S_DO;\n;(C1);S_OD
T_MW_Proc               S_MW_PROC; ;(C0);S_LPAREN;(C1); ;S_VAR; ;(C2);
                        S_RPAREN; ;S_DEFINE;\n;(C3);S_END
T_MW_Funct              S_MW_FUNCT; ;(C0);S_LPAREN;(C1);S_RPAREN; ;S_DEFINE;\n;S_VAR; ;
                        S_LANGLE;(C2);S_RANGLE;S_COLON;\n;(C3);S_SEMICOLON;\n;S_LPAREN;
                        (C4);S_RPAREN;\n;S_END
T_MW_BFunct             S_MW_BFUNCT; ;(C0);S_QUERY;S_LPAREN;(C1);S_RPAREN; ;S_DEFINE;
                        \n;S_VAR; ;S_LANGLE;(C2);S_RANGLE;S_COLON;\n;(C3);S_SEMICOLON;\n;
                        S_LPAREN;(C4);S_RPAREN;\n;S_END
T_Skip                  S_SKIP
T_Foreach_NAS           S_FOREACH; ;S_NAS; ;S_DO;\n;(A,);S_OD
T_Ateach_NAS            S_ATEACH; ;S_NAS; ;S_DO;\n;(A,);S_OD
T_Foreach_Variable      S_FOREACH; ;S_VARIABLE; ;S_DO;\n;(A,);S_OD
T_Foreach_Global_Var    S_FOREACH; ;S_GLOBAL; ;S_VARIABLE; ;S_DO;\n;(A,);S_OD
T_Ateach_Variable       S_ATEACH; ;S_VARIABLE; ;S_DO;\n;(A,);S_OD
T_Ateach_Global_Var     S_ATEACH; ;S_GLOBAL; ;S_VARIABLE; ;S_DO;\n;(A,);S_OD
T_Foreach_Lvalue        S_FOREACH; ;S_LVALUE; ;S_DO;\n;(A,);S_OD
T_Ateach_Lvalue         S_ATEACH; ;S_LVALUE; ;S_DO;\n;(A,);S_OD
T_For_In                S_FOR; ;(C0); ;S_IN; ;(C1); ;S_DO;\n;(C2);S_OD
T_Puthash               (C0);S_FULLSTOP;S_LPAREN;(C1);S_RPAREN; ;S_BECOMES; ;(C2)
T_Print                 S_PRINT;S_LPAREN;(C0);S_RPAREN
T_Prinflush             S_PRINFLUSH;S_LPAREN;(C0);S_RPAREN
```

```
T_Maphash           S_MAPHASH;S_LPAREN;S_QUOTES;(C0);S_QUOTES;S_COMMA;
                    (C1);S_RPAREN
T_Error             S_ERROR;S_LPAREN;(C0);S_RPAREN
T_Stat_Int_One
T_Stat_Int_Any
T_Stat_Val_One
T_Stat_Val_Any
T_Ifmatch2_Stat     S_IFMATCH2; ;S_STATEMENT;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Expn     S_IFMATCH2; ;S_EXPRESSION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Cond     S_IFMATCH2; ;S_CONDITION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Defn     S_IFMATCH2; ;S_DEFINITION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Lvalue   S_IFMATCH2; ;S_LVALUE;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Assign   S_IFMATCH2; ;S_ASSIGN;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Guarded  S_IFMATCH2; ;S_GUARDED;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Action   S_IFMATCH2; ;S_ACTION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Stats    S_IFMATCH2; ;S_STATEMENTS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Expns    S_IFMATCH2; ;S_EXPRESSIONS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Lvalues  S_IFMATCH2; ;S_LVALUES;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Assigns  S_IFMATCH2; ;S_ASSIGNS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch2_Defns    S_IFMATCH2; ;S_DEFINITIONS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Stat      S_IFMATCH; ;S_STATEMENT;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Expn      S_IFMATCH; ;S_EXPRESSION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Cond      S_IFMATCH; ;S_CONDITION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Defn      S_IFMATCH; ;S_DEFINITION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Lvalue    S_IFMATCH; ;S_LVALUE;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Assign    S_IFMATCH; ;S_ASSIGN;\n;(C0);\n;S_THEN;
```

```
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Guarded   S_IFMATCH; ;S_GUARDED;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Action    S_IFMATCH; ;S_ACTION;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Stats     S_IFMATCH; ;S_STATEMENTS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Expns     S_IFMATCH; ;S_EXPRESSIONS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Lvalues   S_IFMATCH; ;S_LVALUES;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Assigns   S_IFMATCH; ;S_ASSIGNS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_Ifmatch_Defns     S_IFMATCH; ;S_DEFINITIONS;\n;(C0);\n;S_THEN;
                    \n;(C1);S_ELSE;\n;(C2);S_ENDMATCH
T_X_Funct_Call      S_PLINK_XF; ;(C0);S_LPAREN;(C1);S_RPAREN
T_MW_Funct_Call     (C0);S_LPAREN;(C1);S_RPAREN
T_Expn_Place        S_EXPN_PLACE
T_Var_Place         S_VAR_PLACE
T_Number            (V)
T_String            C:34;(V);C:34
T_Variable          (V)
T_Primed_Var
T_Sequence          S_LANGLE; ;(A,); ;S_RANGLE
T_Aref              (C0);S_LBRACKET;(C1);S_RBRACKET
T_Sub_Seg           (C0);S_LBRACKET;(C1);S_DOTDOT;(C2);S_RBRACKET
T_Rel_Seg           (C0);S_LBRACKET;(C1);S_COMMA;(C2);S_RBRACKET
T_Final_Seg         (C0);S_LBRACKET;(C1);S_DOTDOT;S_RBRACKET
T_Funct_Call        (C0);S_LPAREN;(C1);S_RPAREN
T_Map               S_MAP;S_LPAREN;C:34;(C0);C:34;S_COMMA;(C1);S_RPAREN
T_Reduce            S_REDUCE;S_LPAREN;C:34;(C0);C:34;S_COMMA;(C1);S_RPAREN
T_Expn_Pat_One      S_PAT_ONE;(V)
T_Expn_Pat_Many     S_PAT_MANY;(V)
T_Expn_Pat_Any      S_PAT_ANY;(V)
T_Plus              S_LPAREN;(A, #S_PLUS# );S_RPAREN
T_Minus             S_LPAREN;(A, #S_MINUS# );S_RPAREN
T_Times             S_LPAREN;(A, #S_TIMES# );S_RPAREN
T_Divide            S_LPAREN;(A, #S_SLASH# );S_RPAREN
T_Exponent          (C0); ;S_EXPONENT; ;(C1)
T_Mod               S_LPAREN;(C0); ;S_MOD; ;(C1);S_RPAREN;
T_Div               S_LPAREN;(C0); ;S_DIV; ;(C1);S_RPAREN;
T_If                S_IF; ;(C0); ;S_THEN; ;(C1); ;S_ELSE; ;(C2); ;S_FI
T_Abs               S_ABS;S_LPAREN;(A,);S_RPAREN
T_Frac              S_FRAC;S_LPAREN;(A,);S_RPAREN
```

```
T_Int                S_INT;S_LPAREN;(A,);S_RPAREN
T_Sgn                S_SGN;S_LPAREN;(A,);S_RPAREN
T_Max                S_MAX;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Min                S_MIN;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Intersection       (C0); ;S_INTERSECT; ;(C1)
T_Union              (C0); ;S_UNION; ;(C1)
T_Set_Diff           (C0); ;S_BACKSLASH; ;(C1)
T_Powerset           S_POWERSET;S_LPAREN;(A,);S_RPAREN
T_Set                S_LBRACE;(C0); ;S_VBAR; ;(C1);S_RBRACE
T_Array              S_ARRAY;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Head               S_HEAD;S_LPAREN;(A,);S_RPAREN
T_Tail               S_TAIL;S_LPAREN;(A,);S_RPAREN
T_Last               S_LAST;S_LPAREN;(A,);S_RPAREN
T_Butlast            S_BUTLAST;S_LPAREN;(A,);S_RPAREN
T_Length             S_LENGTH;S_LPAREN;(A,);S_RPAREN
T_Reverse            S_REVERSE;S_LPAREN;(A,);S_RPAREN
T_Concat             (C0); ;S_CONCAT; ;(C1)
T_Negate             S_MINUS;(C0)
T_Invert
T_Struct             (C1);S_FULLSTOP;(C0)
T_Get_n              (C0);S_CARET;(C1)
T_Get                (C0);S_CARET;S_CARET;(C1)
T_Gethash            (C0);S_FULLSTOP;S_LPAREN;(C1);S_RPAREN
T_Hash_Table         S_HASH_TABLE
T_Slength            S_SLENGTH;S_LPAREN;(A,);S_RPAREN
T_Substr             S_SUBSTR;S_LPAREN;(A,);S_RPAREN
T_Index              S_INDEX;S_LPAREN;(A,);S_RPAREN
T_Expn_Int_One
T_Expn_Int_Any
T_Expn_Val_One
T_Expn_Val_Any
T_Fill2_Stat         S_FILL; ;S_STATEMENT;\n;(C0);\n;S_ENDFILL
T_Fill2_Expn         S_FILL; ;S_EXPRESSION;\n;(C0);\n;S_ENDFILL
T_Fill2_Cond         S_FILL; ;S_CONDITION;\n;(C0);\n;S_ENDFILL
T_Fill2_Defn         S_FILL; ;S_DEFINITION;\n;(C0);\n;S_ENDFILL
T_Fill2_Lvalue       S_FILL; ;S_LVALUE;\n;(C0);\n;S_ENDFILL
T_Fill2_Assign       S_FILL; ;S_ASSIGN;\n;(C0);\n;S_ENDFILL
T_Fill2_Guarded      S_FILL; ;S_GUARDED;\n;(C0);\n;S_ENDFILL
T_Fill2_Action       S_FILL; ;S_ACTION;\n;(C0);\n;S_ENDFILL
T_Fill2_Stats        S_FILL; ;S_STATEMENTS;\n;(C0);\n;S_ENDFILL
T_Fill2_Expns        S_FILL; ;S_EXPRESSIONS;\n;(C0);\n;S_ENDFILL
T_Fill2_Lvalues      S_FILL; ;S_LVALUES;\n;(C0);\n;S_ENDFILL
T_Fill2_Assigns      S_FILL; ;S_ASSIGNS;\n;(C0);\n;S_ENDFILL
T_Fill2_Defns        S_FILL; ;S_DEFINITIONS;\n;(C0);\n;S_ENDFILL
```

```
T_Fill_Stat           S_FILL; ;S_STATEMENT;\n;(C0);\n;S_ENDFILL
T_Fill_Expn           S_FILL; ;S_EXPRESSION;\n;(C0);\n;S_ENDFILL
T_Fill_Cond           S_FILL; ;S_CONDITION;\n;(C0);\n;S_ENDFILL
T_Fill_Defn           S_FILL; ;S_DEFINITION;\n;(C0);\n;S_ENDFILL
T_Fill_Lvalue         S_FILL; ;S_LVALUE;\n;(C0);\n;S_ENDFILL
T_Fill_Assign         S_FILL; ;S_ASSIGN;\n;(C0);\n;S_ENDFILL
T_Fill_Guarded        S_FILL; ;S_GUARDED;\n;(C0);\n;S_ENDFILL
T_Fill_Action         S_FILL; ;S_ACTION;\n;(C0);\n;S_ENDFILL
T_Fill_Stats          S_FILL; ;S_STATEMENTS;\n;(C0);\n;S_ENDFILL
T_Fill_Expns          S_FILL; ;S_EXPRESSIONS;\n;(C0);\n;S_ENDFILL
T_Fill_Lvalues        S_FILL; ;S_LVALUES;\n;(C0);\n;S_ENDFILL
T_Fill_Assigns        S_FILL; ;S_ASSIGNS;\n;(C0);\n;S_ENDFILL
T_Fill_Defns          S_FILL; ;S_DEFINITIONS;\n;(C0);\n;S_ENDFILL
T_X_BFunct_Call       S_PLINK_XC; ;(C0);S_QUERY;S_LPAREN;(C1);S_RPAREN
T_MW_BFunct_Call      (C0);S_QUERY;S_LPAREN;(C1);S_RPAREN
T_Cond_Place          S_COND_PLACE
T_BFunct_Call         (C0);S_QUERY;S_LPAREN;(C1);S_RPAREN
T_Cond_Pat_One        S_PAT_ONE;(V)
T_Cond_Pat_Many       S_PAT_MANY;(V)
T_Cond_Pat_Any        S_PAT_ANY;(V)
T_True                S_TRUE
T_False               S_FALSE
T_And                 (C0); ;S_AND; ;(C1)
T_Or                  (C0); ;S_OR; ;(C1)
T_Not                 S_NOT; ;(A,)
T_Equal               (C0); ;S_EQUAL; ;(C1)
T_Less                (C0); ;S_LANGLE; ;(C1)
T_Greater             (C0); ;S_RANGLE; ;(C1)
T_Less_Eq             (C0); ;S_LEQ; ;(C1)
T_Greater_Eq          (C0); ;S_GEQ; ;(C1)
T_Not_Equal           (C0); ;S_NEQ; ;(C1)
T_Even                S_EVEN;S_QUERY;S_LPAREN;(A,);S_RPAREN
T_Odd                 S_ODD;S_QUERY;S_LPAREN;(A,);S_RPAREN
T_Empty               S_EMPTY;S_QUERY;S_LPAREN;(A,);S_RPAREN
T_Subset              S_SUBSET;S_QUERY;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Member              S_MEMBER;S_QUERY;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Forall              S_FORALL;S_LANGLE;(C0);S_RANGLE;S_COLON; ;(C1); ;S_END
T_Exists              S_EXISTS;S_LANGLE;(C0);S_RANGLE;S_COLON; ;(C1); ;S_END
T_Implies             S_IMPLIES;S_QUERY;S_LPAREN;(C0);S_COMMA;(C1);S_RPAREN
T_Sequenceq
T_Numberq
T_Stringq
T_In                  (C0); ;S_IN; ;(C1)
T_Not_In              (C0); ;S_NOTIN; ;(C1)
```

```
T_Cond_Int_One
T_Cond_Int_Any
T_Cond_Val_One
T_Cond_Val_Any
T_Proc              S_PROC; ;(C0);S_LPAREN;(C1); ;S_VAR; ;(C2);S_RPAREN; ;S_DEFINE;
                    \n;(C3);S_END
T_Funct             S_FUNCT; ;(C0);S_LPAREN;(C1);S_RPAREN; ;S_DEFINE;\n;S_VAR; ;
                    S_LANGLE;(C2);S_RANGLE;S_COLON;\n;S_LPAREN;(C3);S_RPAREN;\n;S_END
T_BFunct            S_BFUNCT; ;(C0);S_QUERY;S_LPAREN;(C1);S_RPAREN; ;S_DEFINE;\n;
                    S_VAR; ;S_LANGLE;(C2);S_RANGLE;S_COLON;\n;S_LPAREN;(C3);S_RPAREN;
                    \n;S_END
T_Defn_Pat_One      S_PAT_ONE;(V)
T_Defn_Pat_Many     S_PAT_MANY;(V)
T_Defn_Pat_Any      S_PAT_ANY;(V)
T_Defn_Int_One
T_Defn_Int_Any
T_Defn_Val_One
T_Defn_Val_Any
T_Var_Lvalue        (V)
T_Aref_Lvalue       (C0);S_LBRACKET;(C1);S_RBRACKET
T_Sub_Seg_Lvalue    (C0);S_LBRACKET;(C1);S_DOTDOT;(C2);S_RBRACKET
T_Rel_Seg_Lvalue    (C0);S_LBRACKET;(C1);S_COMMA;(C2);S_RBRACKET
T_Final_Seg_Lvalue  (C0);S_LBRACKET;(C1);S_DOTDOT;S_RBRACKET
T_Lvalue_Pat_One    S_PAT_ONE;(V)
T_Lvalue_Pat_Many   S_PAT_MANY;(V)
T_Lvalue_Pat_Any    S_PAT_ANY;(V)
T_Struct_Lvalue     (C1);S_FULLSTOP;(C0)
T_Lvalue_Int_One
T_Lvalue_Int_Any
T_Lvalue_Val_One
T_Lvalue_Val_Any
T_Assign_Pat_One    S_PAT_ONE;(V)
T_Assign_Pat_Any    S_PAT_ANY;(V)
T_Assign_Pat_Many   S_PAT_MANY;(V)
T_Assign_Int_One
T_Assign_Int_Any
T_Assign_Val_One
T_Assign_Val_Any
T_Guarded_Pat_One   S_PAT_ONE;(V)
T_Guarded_Pat_Any   S_PAT_ANY;(V)
T_Guarded_Pat_Many  S_PAT_MANY;(V)
T_Guarded_Int_One
T_Guarded_Int_Any
T_Guarded_Val_One
```

```
T_Guarded_Val_Any
T_Action_Pat_One        S_PAT_ONE;(V)
T_Action_Pat_Any        S_PAT_ANY;(V)
T_Action_Pat_Many       S_PAT_MANY;(V)
T_Action_Int_One
T_Action_Int_Any
T_Action_Val_One
T_Action_Val_Any
T_Name_Pat_One          S_PAT_ONE;(V)
T_Name_Int_One
T_Name_Val_One
```

B.5 JavaCC Definition for Untyped WSL

```
/*
 *  The WSL definition for JavaCC
 *
 *  by Matthias Ladkau (matthias@ladkau.de)
 */


options {
  MULTI=true;
  LOOKAHEAD=3;
  STATIC=false;
  DEBUG_PARSER=false;
  DEBUG_LOOKAHEAD=false;
  DEBUG_TOKEN_MANAGER=false;
}

PARSER_BEGIN(WSLParser)
package dynamic_typed_wsl;

import java.io.InputStream;

public class WSLParser {
  public static void main(String args[]) {
    System.out.println("Reading from standard input...");
    WSLParser p = new WSLParser(System.in);
    try {
      ASTparseWSL n = p.parseWSL();
      n.dump("");
      System.out.println("Parsing was successful");
    } catch (Exception e) {
      System.out.println("Got an error:");
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }

  public static ASTparseWSL parseStream(InputStream stream) throws Exception {
    WSLParser p = new WSLParser(stream);
    return p.parseWSL();
  }
}
```

```
PARSER_END(WSLParser)


/* ========
    Lexer Part
    ========
 */


SKIP : /* Skip WhiteSpace */
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN : /* KEYWORDS */
{
  < S_ABORT: "ABORT" >
| < S_ABS: "ABS" >
| < S_ACTION: "Action" >
| < S_ACTIONS: "ACTIONS" >
| < S_AND: "AND" >
| < S_ARRAY: "ARRAY" >
| < S_ASSIGN: "Assign" >
| < S_ASSIGNS: "Assigns" >
| < S_ATEACH: "ATEACH" >
| < S_BEGIN: "BEGIN" >
| < S_BFUNCT: "BFUNCT" >
| < S_BUTLAST: "BUTLAST" >
| < S_CALL: "CALL" >
| < S_COMMENT: "Comment:" | "COMMENT:" | "C:">
| < S_COND_PLACE: "$Condition$" >
| < S_CONDITION: "Condition" >
| < S_D_DO: "D_DO" >
| < S_D_IF: "D_IF" >
| < S_DEFINITION: "Definition" >
| < S_DEFINITIONS: "Definitions" >
| < S_DIV: "DIV" >
| < S_DO: "DO" >
| < S_ELSE: "ELSE" >
| < S_ELSIF: "ELSIF" >
| < S_EMPTY: "EMPTY" >
| < S_END: "END" >
```

```
| < S_ENDACTIONS: "ENDACTIONS" >
| < S_ENDFILL: "ENDFILL" >
| < S_ENDJOIN: "ENDJOIN" >
| < S_ENDMATCH: "ENDMATCH" >
| < S_ENDSPEC: "ENDSPEC" >
| < S_ENDVAR: "ENDVAR" >
| < S_ERROR: "ERROR" >
| < S_EVEN: "EVEN" >
| < S_EXISTS: "EXISTS" >
| < S_EXIT: "EXIT" >
| < S_EXPN_PLACE: "$Expn$" >
| < S_EXPRESSION: "Expression" >
| < S_EXPRESSIONS: "Expressions" >
| < S_FALSE: "FALSE" >
| < S_FI: "FI" >
| < S_FILL: "FILL" >
| < S_FILL2: "FILL2" >
| < S_FOR: "FOR" >
| < S_FORALL: "FORALL" >
| < S_FOREACH: "FOREACH" >
| < S_FRAC: "FRAC" >
| < S_FUNCT: "FUNCT" >
| < S_GLOBAL: "Global" >
| < S_GUARDED: "Guarded" >
| < S_HASH_TABLE: "HASH_TABLE" >
| < S_HEAD: "HEAD" >
| < S_IF: "IF" >
| < S_IFMATCH: "IFMATCH" >
| < S_IFMATCH2: "IFMATCH2" >
| < S_IN: "IN" >
| < S_INDEX: "INDEX" >
| < S_INT: "INT" >
| < S_INTS: "%Z" >
| < S_JOIN: "JOIN" >
| < S_LAST: "LAST" >
| < S_LENGTH: "LENGTH" >
| < S_LVALUE: "Lvalue" >
| < S_LVALUES: "Lvalues" >
| < S_MAP: "MAP" >
| < S_MAPHASH: "MAPHASH" >
| < S_MAX: "MAX" >
| < S_MEMBER: "MEMBER" >
| < S_MIN: "MIN" >
| < S_MOD: "MOD" >
| < S_MW_BFUNCT: "MW_BFUNCT" >
```

```
| < S_MW_FUNCT: "MW_FUNCT" >
| < S_MW_PROC: "MW_PROC" >
| < S_NAS: "NAS" >
| < S_NATS: "%N" >
| < S_NOT: "NOT" >
| < S_NOTIN: "NOTIN" >
| < S_NUMBERQ: "NUMBER" >
| < S_OD: "OD" >
| < S_ODD: "ODD" >
| < S_OR: "OR" >
| < S_POP: "POP" >
| < S_POWERSET: "POWERSET" >
| < S_PRINFLUSH: "PRINFLUSH" >
| < S_PRINT: "PRINT" >
| < S_PROC: "PROC" >
| < S_PUSH: "PUSH" >
| < S_RATS: "%Q" >
| < S_REALS: "%R" >
| < S_REDUCE: "REDUCE" >
| < S_RETURNS: "RETURNS" >
| < S_REVERSE: "REVERSE" >
| < S_SEQUENCE: "SEQUENCE" >
| < S_SGN: "SGN" >
| < S_SKIP: "SKIP" >
| < S_SLENGTH: "SLENGTH" >
| < S_SPEC: "SPEC" >
| < S_STAT_PLACE: "$Statement$" >
| < S_STATEMENT: "Statement" >
| < S_STATEMENTS: "Statements" >
| < S_STEP: "STEP" >
| < S_STRINGBF: "STRING" >
| < S_STS: "STS" >
| < S_SUBSET: "SUBSET" >
| < S_SUBSTR: "SUBSTR" >
| < S_TAIL: "TAIL" >
| < S_TERMINAL: "Terminal" >
| < S_THEN: "THEN" >
| < S_TO: "TO" >
| < S_TRUE: "TRUE" >
| < S_VAR: "VAR" >
| < S_VAR_PLACE: "$Var$" >
| < S_VARIABLE: "Variable" >
| < S_WHERE: "WHERE" >
| < S_WHILE: "WHILE" >
}
```

```
TOKEN : /* Special Tokens */
{
  < S_ARROW: "->" >
| < S_AT: "@" >
| < S_AT_PAT_ONE: "@~?" >
| < S_BACKSLASH: "\\" >
| < S_BECOMES: ":=" >
| < S_BOX: "[]" >
| < S_CARET: "^" >
| < S_COLON: ":" >
| < S_COMMA: "," >
| < S_CONCAT: "++" >
| < S_DEFINE: "==" >
| < S_DOTDOT: ".." >
| < S_DOTSPACE: ". " >
| < S_EQUAL: "=" >
| < S_EXPONENT: "**" >
| < S_FULLSTOP: "." >
| < S_GEQ: ">=" >
| < S_INTERSECT: "/\\" >
| < S_LANGLE: "<" >
| < S_LBRACE: "{" >
| < S_LBRACKET: "[" >
| < S_LEQ: "<=" >
| < S_LPAREN: "(" >
| < S_MINUS: "-" >
| < S_NEQ: "<>" >
| < S_PAT_ANY: "~*" >
| < S_PAT_MANY: "~+" >
| < S_PAT_ONE: "~?" >
| < S_PLINK_P: "!P" >
| < S_PLINK_XC: "!XC" >
| < S_PLINK_XF: "!XF" >
| < S_PLINK_XP: "!XP" >
| < S_PLUS: "+" >
| < S_PRIME: "'" >
| < S_QUERY: "?" >
| < S_QUOTES: "\"" >
| < S_RANGLE: ">" >
| < S_RBRACE: "}" >
| < S_RBRACKET: "]" >
| < S_RPAREN: ")" >
| < S_SEMICOLON: ";" >
| < S_SLASH: "/" >
```

```
| < S_TIMES: "*" >
| < S_UNION: "\\/" >
| < S_VBAR: "|" >
}

TOKEN : /* Values */
{
  < S_NUMBER: ["0"-"9"] (["0"-"9"])* >

| < S_IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|   < #LETTER: ["_","a"-"z","A"-"Z"] >
|   < #DIGIT: ["0"-"9"] >

| < S_AMBIGOUS_IDENTIFIER: "@JOIN">

| < S_STRING:
      "\""
      ((~["\""])
        | (["_","a"-"z","A"-"Z"])
        | (["0"-"9"])
        | "\n"
        | "\r\n"
      )*
      "\""
  >
}

/* ========
    Parser Part
    ========
 */


ASTparseWSL parseWSL() : {}
{
  statements() <EOF>

/*
   Entry points for general nodes

   T_Statements -> statements()
   T_Statement  -> statement()
   T_Expression -> expression()
   T_Expressions        -> expressions()
   T_Condition  -> condition()
```

```
   T_Definition -> define()
   T_Definitions        -> defines()
   T_Assign             -> assign()
   T_Assigns    -> assigns_node()
   T_Action             -> action()
   T_Guarded    -> guarded()
   T_Lvalue             -> lvalue()
   T_Lvalues    -> lvalues()
*/

  { return jjtThis; }
}

/*
 * Statements
 * ==========
 */

void statements() #T_Statements : {}
{
  statement() (<S_SEMICOLON> statement())*
}



void statement() #void : {}
{
  stat_if()
| stat_d_if()
| stat_d_do()
| stat_while()
| stat_do()
| stat_exit()
| stat_for()
| stat_var()
| stat_comment()
| stat_assert()
| stat_assignment()
| stat_push()
| stat_pop()
| stat_join()
| stat_actions()
| stat_call()
| stat_print()
| stat_mw_func_decl()
```

```
| stat_begin()
| stat_foreach()
| stat_ateach()
| stat_ifmatch()
| stat_ifmatch2()
| stat_maphash()
| stat_error()
| stat_spec()

/* Assignments */
| stat_single_assign()

/* Pattern Statements */
| stat_pattern()

/* Proc Call */
| stat_proc_call()

/* Trivial Statements */
| (<S_SKIP>)#T_Skip
| (<S_ABORT>)#T_Abort
| (<S_STAT_PLACE>)#T_Stat_Place
}

void stat_if() #void  : {}
{
  (
    (<S_IF> condition() <S_THEN> statements())#T_Guarded(2)
    ((<S_ELSIF> condition() <S_THEN> statements())#T_Guarded(2))*
    (
      (<S_ELSE>#T_True statements() <S_FI>)#T_Guarded (2)
|     pseudo_else() <S_FI>
    )
  )#T_Cond
}

void stat_d_if() #void  : {}
{
  (
    (<S_D_IF> condition() <S_ARROW> statements())#T_Guarded(2)
    ((<S_BOX> condition() <S_ARROW> statements())#T_Guarded(2))*
    <S_FI>
  )#T_D_If
}
```

```
void stat_d_do() #void  : {}
{
  (
    (<S_D_DO> condition() <S_ARROW> statements())#T_Guarded(2)
    ((<S_BOX> condition() <S_ARROW> statements())#T_Guarded(2))*
    <S_OD>
  )#T_D_Do
}

void stat_while() #void  : {}
{
  (<S_WHILE> condition() <S_DO> statements() <S_OD>)#T_While
}

void stat_do() #void  : {}
{
  (<S_DO> statements() <S_OD>)#T_Floop
}

void stat_exit() #void  : {}
{
  T_Exit()
}

void stat_for() #void  : {}
{
  (
    <S_FOR> T_Var_Lvalue() <S_BECOMES> expression()
    <S_TO> expression()
    <S_STEP> expression()
    <S_DO> statements() <S_OD>
  )#T_For

| (
    <S_FOR> T_Var_Lvalue()
    <S_IN> s_expression()
    <S_DO> statements() <S_OD>
  )#T_For_In
}

void stat_var() #void  : {}
{
  (<S_VAR> <S_LANGLE> assigns()#T_Assigns <S_RANGLE> <S_COLON> statements() <
      S_ENDVAR>)#T_Var
}
```

```
void stat_comment() #void  : {}
{
  T_Comment()
}

void stat_assert() #void  : {}
{
  (<S_LBRACE> condition() <S_RBRACE>)#T_Assert
}

void stat_assignment() #void  : {}
{
  (<S_LANGLE> assigns() <S_RANGLE>)#T_Assignment
}

void stat_push() #void  : {}
{
  (<S_PUSH> <S_LPAREN> T_Var_Lvalue() <S_COMMA> s_expression() <S_RPAREN>)#
      T_Push
}

void stat_pop() #void  : {}
{
  (<S_POP>  <S_LPAREN> T_Var_Lvalue() <S_COMMA> T_Var_Lvalue() <S_RPAREN>)#
      T_Pop
}

void stat_join() #void  : {}
{
  (<S_JOIN>  statements() <S_COMMA> statements() <S_ENDJOIN>)#T_Join
}

void stat_actions() #void  : {}
{
  (<S_ACTIONS> T_IdentifierName() <S_COLON> actions() <S_ENDACTIONS>)#T_A_S
}

void stat_call() #void  : {}
{
  T_Call()
}

void stat_print() #void  : {}
{
```

```
  (<S_PRINT> <S_LPAREN> expressions() <S_RPAREN>)#T_Print
| (<S_PRINFLUSH> <S_LPAREN> expressions() <S_RPAREN>)#T_Prinflush
}

void stat_mw_func_decl() #void  : {}
{
  (<S_MW_PROC> T_AtName() <S_LPAREN>  ((lvalue() (<S_COMMA> lvalue())*)*)#
      T_Lvalues var_lvalues() <S_RPAREN>
    <S_DEFINE> statements()
    (<S_END> | <S_FULLSTOP>))#T_MW_Proc

|  (<S_MW_FUNCT> T_AtName() <S_LPAREN>  ((lvalue() (<S_COMMA> lvalue())*)*)#
    T_Lvalues <S_RPAREN>
    <S_DEFINE>
     (
        (<S_VAR> <S_LANGLE> assigns()#T_Assigns <S_RANGLE> <S_COLON>
        statements() <S_SEMICOLON>
        <S_LPAREN> expression() <S_RPAREN>
        (<S_END> | <S_FULLSTOP>))
|
        (<S_COLON>#T_Assigns statements() <S_SEMICOLON>
        <S_LPAREN> expression() <S_RPAREN>
        (<S_END> | <S_FULLSTOP>))
     )
   )#T_MW_Funct


| (<S_MW_BFUNCT> T_AtName() <S_QUERY> <S_LPAREN>  ((lvalue() (<S_COMMA> lvalue
    ())*)*)#T_Lvalues <S_RPAREN>
    <S_DEFINE>
    (
      (<S_VAR> <S_LANGLE> assigns()#T_Assigns <S_RANGLE> <S_COLON>
      statements() <S_SEMICOLON>
      <S_LPAREN> condition() <S_RPAREN>
      (<S_END> | <S_FULLSTOP>))
|
      (<S_COLON>#T_Assigns statements() <S_SEMICOLON>
      <S_LPAREN> condition() <S_RPAREN>
      (<S_END> | <S_FULLSTOP>))
    )
  )#T_MW_BFunct

}

void stat_begin() #void  : {}
```

```
{
  (<S_BEGIN>
  statements()
  <S_WHERE>
  defines()
  <S_END>)#T_Where
}

void stat_foreach() #void : {}
{
  <S_FOREACH>
  (
    (<S_STATEMENT> <S_DO> statements() <S_OD>)#T_Foreach_Stat
|   (<S_STATEMENTS> <S_DO> statements() <S_OD>)#T_Foreach_Stats
|   (<S_VARIABLE> <S_DO> statements() <S_OD>)#T_Foreach_Variable
|   (<S_GLOBAL> <S_VARIABLE> <S_DO> statements() <S_OD>)#T_Foreach_Global_Var
|   (<S_LVALUE> <S_DO> statements() <S_OD>)#T_Foreach_Lvalue
|   (<S_STS> <S_DO> statements() <S_OD>)#T_Foreach_STS
|   (<S_NAS> <S_DO> statements() <S_OD>)#T_Foreach_NAS
|   (<S_EXPRESSION> <S_DO> statements() <S_OD>)#T_Foreach_Expn
|   (<S_CONDITION> <S_DO> statements() <S_OD>)#T_Foreach_Cond
|   (<S_TERMINAL>
      (
        (<S_STATEMENT> <S_DO> statements() <S_OD>)#T_Foreach_TS
|       (<S_STATEMENTS> <S_DO> statements() <S_OD>) #T_Foreach_TSs
      )
    )
  )
}

void stat_ateach() #void : {}
{
  <S_ATEACH>
  (
    (<S_STATEMENT> <S_DO> statements() <S_OD>)#T_Ateach_Stat
|   (<S_STATEMENTS> <S_DO> statements() <S_OD>)#T_Ateach_Stats
|   (<S_VARIABLE> <S_DO> statements() <S_OD>)#T_Ateach_Variable
|   (<S_GLOBAL> <S_VARIABLE> <S_DO> statements() <S_OD>)#T_Ateach_Global_Var
|   (<S_LVALUE> <S_DO> statements() <S_OD>)#T_Ateach_Lvalue
|   (<S_STS> <S_DO> statements() <S_OD>)#T_Ateach_STS
|   (<S_NAS> <S_DO> statements() <S_OD>)#T_Ateach_NAS
|   (<S_EXPRESSION> <S_DO> statements() <S_OD>)#T_Ateach_Expn
|   (<S_CONDITION> <S_DO> statements() <S_OD>)#T_Ateach_Cond
|   (<S_TERMINAL>
      (
```

```
        (<S_STATEMENT> <S_DO> statements() <S_OD>)#T_Ateach_TS
|       (<S_STATEMENTS> <S_DO> statements() <S_OD>) #T_Ateach_TSs
      )
    )
  )
}

void stat_ifmatch() #void : {}
{
  <S_IFMATCH>
  (
    (<S_STATEMENTS> statements() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)
        #T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Stats
|   (<S_STATEMENT> statement() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Stat
|   (<S_EXPRESSION> expression() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)
    #T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Expn
|   (<S_EXPRESSIONS> expressions() <S_THEN> statements() (((<S_ENDMATCH>)#
    T_Skip)#T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #
    T_Ifmatch_Expns
|   (<S_CONDITION> condition() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Cond
|   (<S_DEFINITION> define() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Defn
|   (<S_DEFINITIONS> defines() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Defns
|   (<S_ASSIGN> assign() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Assign
|   (<S_ASSIGNS> assigns_node() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Assigns
|   (<S_ACTION> action() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Action
|   (<S_GUARDED> guarded() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Guarded
|   (<S_LVALUE> lvalue() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Lvalue
|   (<S_LVALUES> lvalues() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch_Lvalues
  )
}

void stat_ifmatch2() #void : {}
{
  <S_IFMATCH2>
  (
```

```
    (<S_STATEMENTS> statements() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)
        #T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #
        T_Ifmatch2_Stats
|   (<S_STATEMENT> statement() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Stat
|   (<S_EXPRESSION> expression() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)
    #T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Expn
|   (<S_EXPRESSIONS> expressions() <S_THEN> statements() (((<S_ENDMATCH>)#
    T_Skip)#T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #
    T_Ifmatch2_Expns
|   (<S_CONDITION> condition() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Cond
|   (<S_DEFINITION> define() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Defn
|   (<S_DEFINITIONS> defines() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Defns
|   (<S_ASSIGN> assign() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Assign
|   (<S_ASSIGNS> assigns_node() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Assigns
|   (<S_ACTION> action() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Action
|   (<S_GUARDED> guarded() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Guarded
|   (<S_LVALUE> lvalue() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Lvalue
|   (<S_LVALUES> lvalues() <S_THEN> statements() (((<S_ENDMATCH>)#T_Skip)#
    T_Statements | (<S_ELSE> statements() <S_ENDMATCH>))) #T_Ifmatch2_Lvalues
  )
}

void stat_maphash() #void : {}
{
  (<S_MAPHASH> <S_LPAREN> T_Name() <S_COMMA> expression() <S_RPAREN>)#T_Maphash
}

void stat_error() #void : {}
{
  (<S_ERROR> <S_LPAREN> expressions() <S_RPAREN>)#T_Error
}


void stat_spec() #void : {}
{
```

```
  (<S_SPEC> <S_LANGLE> lvalues() <S_RANGLE> <S_COLON> condition() <S_ENDSPEC>)#
      T_Spec
}

void stat_proc_call() #void  : {}
{
(T_IdentifierName() <S_LPAREN> ((expression() (<S_COMMA> expression())*)*)#
    T_Expressions var_lvalues() <S_RPAREN>)#T_Proc_Call
| (<S_PLINK_P> T_IdentifierName() <S_LPAREN> ((expression() (<S_COMMA>
    expression())*)*)#T_Expressions var_lvalues() <S_RPAREN>)#T_A_Proc_Call
| (T_AtName() (<S_LPAREN>)* ((expression() (<S_COMMA> expression())*)*)#
    T_Expressions var_lvalues() (<S_RPAREN>)*)#T_MW_Proc_Call
| (T_AtPatOneName() (<S_LPAREN>)* ((expression() (<S_COMMA> expression())*)*)#
    T_Expressions var_lvalues() (<S_RPAREN>)*)#T_MW_Proc_Call
| (<S_PLINK_XP> T_IdentifierName() <S_LPAREN> ((expression() (<S_COMMA>
    expression())*)*)#T_Expressions <S_RPAREN>)#T_X_Proc_Call
}

void stat_pattern() #void  : {}
{
  T_Stat_Pat_One()
| T_Stat_Pat_Many()
| T_Stat_Pat_Any()

}

void stat_single_assign() #void  : {}
{
  ((lvalue() <S_BECOMES> expression())#T_Assign)#T_Assignment
| (lvalue() <S_FULLSTOP> <S_LPAREN> expression() <S_RPAREN> <S_BECOMES>
    expression())#T_Puthash
}


/*
 * Guarded
 * ======
 */

void guarded() #void  : {}
{
  (condition() (<S_THEN> | <S_ARROW>) statements())#T_Guarded(2)
| (
    (
  /* Pattern Expressions */
```

```
      T_Cond_Pat_One()
|     T_Cond_Pat_Many()
|     T_Cond_Pat_Any()
    )
    (<S_THEN> | <S_ARROW>) statements()
  )#T_Guarded(2)
}


/*
 * Defines
 * ======
 */

void defines() #void  : {}
{
  (define()
  (
    (define())
|   (<S_COMMA> define())
  )*
  )#T_Definitions
}

void define() #void  : {}
{
    stat_func_decl()

  /* Pattern Expressions */
|     T_Defn_Pat_One()
|     T_Defn_Pat_Many()
|     T_Defn_Pat_Any()
}

void stat_func_decl() #void : {}
{
  (<S_PROC> T_IdentifierName() <S_LPAREN>  ((lvalue() (<S_COMMA> lvalue())*)*)#
      T_Lvalues var_lvalues() <S_RPAREN>
    <S_DEFINE> statements()
    (<S_END> | <S_FULLSTOP>))#T_Proc

| (<S_FUNCT> T_IdentifierName() <S_LPAREN>  ((lvalue() (<S_COMMA> lvalue())*)*)
    #T_Lvalues <S_RPAREN>
    <S_DEFINE>
    (
```

```
      (<S_VAR> <S_LANGLE> ((assigns())*)#T_Assigns <S_RANGLE> <S_COLON>
      <S_LPAREN> expression() <S_RPAREN>
      (<S_END> | <S_FULLSTOP>))
|
      (<S_COLON>#T_Assigns <S_LPAREN> expression() <S_RPAREN>
      (<S_END> | <S_FULLSTOP>))
    )
  )#T_Funct

| (<S_BFUNCT> T_IdentifierName() <S_QUERY> <S_LPAREN>  ((lvalue() (<S_COMMA>
    lvalue())*)*)#T_Lvalues <S_RPAREN>
    <S_DEFINE>
    (
      (<S_VAR> <S_LANGLE> ((assigns())*)#T_Assigns <S_RANGLE> <S_COLON>
      <S_LPAREN> condition() <S_RPAREN>
      (<S_END> | <S_FULLSTOP>))
|
      (<S_COLON>#T_Assigns <S_LPAREN> condition() <S_RPAREN>
      (<S_END> | <S_FULLSTOP>))
    )
  )#T_BFunct
}


/*
 * Action System
 * ============
 */
void actions() #void  : {}
{
  (action() (action())*)#T_Actions
}

void action() #void  : {}
{
  (
    (
      T_IdentifierName()

/* Pattern Expressions */
|     T_Action_Pat_One()
|     T_Action_Pat_Many()
|     T_Action_Pat_Any()
    )
  <S_DEFINE> statements() (<S_END> | <S_DOTSPACE>))#T_Action
```

```
}


/*
 * Assignments
 * ==========
 */

void assigns_node() #void  : {}
{
  (assign() (<S_COMMA> assign())*)#T_Assigns
}

void assigns() #void  : {}
{
  assign() (<S_COMMA> assign())*
}

void assign() #void  : {}
{
  (lvalue() <S_BECOMES> expression())#T_Assign

/* Pattern Expressions */
| T_Assign_Pat_One()
| T_Assign_Pat_Many()
| T_Assign_Pat_Any()
}


/*
 * Lvalues
 * ======
 */

void var_lvalues() #void  : {}
{
  ((<S_VAR> (lvalue() (<S_COMMA> lvalue())*)*)*)#T_Lvalues
}

void lvalues() #void  : {}
{
  (lvalue() (<S_COMMA> lvalue())*)#T_Lvalues
}

void lvalue() #void  : {}
```

```
{
(
  T_Var_Lvalue()

/* Pattern Expressions */
| T_Lvalue_Pat_One()
| T_Lvalue_Pat_Many()
| T_Lvalue_Pat_Any()
)

  (
/*Array ref*/
  <S_LBRACKET> a_expressions() (
    <S_RBRACKET>#T_Aref_Lvalue(2)
|   <S_DOTDOT> (
      <S_RBRACKET>#T_Final_Seg_Lvalue(2)
|     a_expression() <S_RBRACKET>#T_Sub_Seg_Lvalue(3)
    )
|   <S_COMMA> a_expression() <S_RBRACKET>#T_Rel_Seg_Lvalue(3)
  )



/* Struct Lvalue */
|   (T_Struct_Lvalue())
  )*
}


/*
 * Conditions
 * ========
 */

void condition() #void : {}
{
  (b_term() (<S_OR> b_term())*)#T_Or(>1)
}

void b_term() #void : {}
{
  (b_factor() (<S_AND> b_factor())*)#T_And(>1)
}

void b_factor() #void : {}
```

```
{
  (<S_NOT> b_factor())#T_Not
| cond_prefix()
| b_atom()
}

void b_atom() #void : {}
{
 (<S_TRUE>)#T_True
| (<S_FALSE>)#T_False
| (<S_COND_PLACE>)#T_Cond_Place
| cond_pat()
| rel_exp()
}

void cond_pat() #void : {}
{

/* Expression Pattern */
  (
  T_Expn_Pat_One()
| T_Expn_Pat_Many()
| T_Expn_Pat_Any()
  )
  (
    <S_EXPONENT> factor()#T_Exponent(2)
|   <S_TIMES> factor()#T_Times(2)
|   <S_SLASH> factor()#T_Divide(2)
|   <S_MOD> factor()#T_Mod(2)
|   <S_DIV> factor()#T_Div(2)
/* Set Op */
|   <S_BACKSLASH> factor()#T_Set_Diff(2)

|   <S_PLUS> term()#T_Plus(2)
|   <S_MINUS> term()#T_Minus(2)
/* Set Op */
|   <S_CONCAT> term()#T_Concat(2)
|   <S_UNION> term()#T_Union(2)
|   <S_INTERSECT> term()#T_Intersection(2)

|   <S_EQUAL> expression()#T_Equal(2)
|   <S_NEQ> expression()#T_Not_Equal(2)
|   <S_LANGLE> expression()#T_Less(2)
|   <S_RANGLE> expression()#T_Greater(2)
|   <S_LEQ> expression()#T_Less_Eq(2)
```

```
|   <S_GEQ> expression()#T_Greater_Eq(2)
|   <S_IN> expression()#T_In(2)
|   <S_NOTIN> expression()#T_Not_In(2)
  )+

| T_Cond_Pat_One()
| T_Cond_Pat_Many()
| T_Cond_Pat_Any()
}

void rel_exp() #void : {}
{
  expression() (
    <S_EQUAL> expression()#T_Equal(2)
|   <S_NEQ> expression()#T_Not_Equal(2)
|   <S_LANGLE> expression()#T_Less(2)
|   <S_RANGLE> expression()#T_Greater(2)
|   <S_LEQ> expression()#T_Less_Eq(2)
|   <S_GEQ> expression()#T_Greater_Eq(2)
|   <S_IN> expression()#T_In(2)
|   <S_NOTIN> expression()#T_Not_In(2)
  )*
}

void cond_prefix() #void : {}
{
  (<S_EVEN> <S_QUERY> <S_LPAREN> expression() <S_RPAREN>)#T_Even
| (<S_ODD> <S_QUERY> <S_LPAREN> expression() <S_RPAREN>)#T_Odd
| (<S_SEQUENCE> <S_QUERY> <S_LPAREN> expression() <S_RPAREN>)#T_Sequenceq
| (<S_NUMBERQ> <S_QUERY> <S_LPAREN> expression() <S_RPAREN>)#T_Numberq
| (<S_STRINGBF> <S_QUERY> <S_LPAREN> expression() <S_RPAREN>)#T_Stringq
| (<S_EMPTY> <S_QUERY> <S_LPAREN> expression() <S_RPAREN>)#T_Empty
| (<S_SUBSET> <S_QUERY> <S_LPAREN> s_expression() <S_COMMA> s_expression() <
    S_RPAREN>)#T_Subset
| (<S_MEMBER> <S_QUERY> <S_LPAREN> expression() <S_COMMA> s_expression() <
    S_RPAREN>)#T_Member
| (<S_FORALL> <S_LANGLE> lvalues() <S_RANGLE> <S_COLON> condition() <S_END>)#
    T_Forall
| (<S_EXISTS> <S_LANGLE> lvalues() <S_RANGLE> <S_COLON> condition() <S_END>)#
    T_Exists
| (T_IdentifierName() <S_QUERY> <S_LPAREN> ((expression() (<S_COMMA> expression
    ())*)*)#T_Expressions <S_RPAREN>)#T_BFunct_Call
| (T_AtName() <S_QUERY> ((<S_LPAREN> expression() (<S_COMMA> expression())* <
    S_RPAREN>)*)#T_Expressions)#T_MW_BFunct_Call
| (<S_PLINK_XC> T_IdentifierName() (
```

```
    (<S_QUERY> <S_LPAREN>)
|   (<S_LPAREN>)
  )
  ((expression() (<S_COMMA> expression())*)*)#T_Expressions <S_RPAREN>)#
      T_X_BFunct_Call

}


/*
 * Expressions
 * =========
 */

void expressions() #void : {}
{
  (expression() (<S_COMMA> expression())*)#T_Expressions
}

void expression() #void : {}
{
  a_expression()
| fill_expression()
| fill2_expression()
| <S_HASH_TABLE>#T_Hash_Table
}


/* If expression */

void if_expression() #void : {}
{
  (<S_IF> condition() <S_THEN> expression() <S_ELSE> expression() <S_FI>)#T_If
}


/* Fill / Fill2 expressions */

void fill_expression() #void : {}
{
  (<S_FILL> <S_STATEMENTS> statements() <S_ENDFILL>)#T_Fill_Stats
| (<S_FILL> <S_STATEMENT> statement() <S_ENDFILL>)#T_Fill_Stat
| (<S_FILL> <S_EXPRESSION> expression() <S_ENDFILL>)#T_Fill_Expn
| (<S_FILL> <S_EXPRESSIONS> expressions() <S_ENDFILL>)#T_Fill_Expns
| (<S_FILL> <S_CONDITION> condition() <S_ENDFILL>)#T_Fill_Cond
```

```
| (<S_FILL> <S_DEFINITION> define() <S_ENDFILL>)#T_Fill_Defn
| (<S_FILL> <S_DEFINITIONS> defines() <S_ENDFILL>)#T_Fill_Defns
| (<S_FILL> <S_ASSIGN> assign() <S_ENDFILL>)#T_Fill_Assign
| (<S_FILL> <S_ASSIGNS> assigns_node() <S_ENDFILL>)#T_Fill_Assigns
| (<S_FILL> <S_ACTION> action() <S_ENDFILL>)#T_Fill_Action
| (<S_FILL> <S_GUARDED> guarded() <S_ENDFILL>)#T_Fill_Guarded
| (<S_FILL> <S_LVALUE> lvalue() <S_ENDFILL>)#T_Fill_Lvalue
| (<S_FILL> <S_LVALUES> lvalues() <S_ENDFILL>)#T_Fill_Lvalues
}

void fill2_expression() #void : {}
{
  (<S_FILL2> <S_STATEMENTS> statements() <S_ENDFILL>)#T_Fill2_Stats
| (<S_FILL2> <S_STATEMENT> statement() <S_ENDFILL>)#T_Fill2_Stat
| (<S_FILL2> <S_EXPRESSION> expression() <S_ENDFILL>)#T_Fill2_Expn
| (<S_FILL2> <S_EXPRESSIONS> expressions() <S_ENDFILL>)#T_Fill2_Expns
| (<S_FILL2> <S_CONDITION> condition() <S_ENDFILL>)#T_Fill2_Cond
| (<S_FILL2> <S_DEFINITION> defines() <S_ENDFILL>)#T_Fill2_Defn
| (<S_FILL2> <S_DEFINITIONS> defines() <S_ENDFILL>)#T_Fill2_Defns
| (<S_FILL2> <S_ASSIGN> assign() <S_ENDFILL>)#T_Fill2_Assign
| (<S_FILL2> <S_ASSIGNS> assigns_node() <S_ENDFILL>)#T_Fill2_Assigns
| (<S_FILL2> <S_ACTION> action() <S_ENDFILL>)#T_Fill2_Action
| (<S_FILL2> <S_GUARDED> guarded() <S_ENDFILL>)#T_Fill2_Guarded
| (<S_FILL2> <S_LVALUE> lvalue() <S_ENDFILL>)#T_Fill2_Lvalue
| (<S_FILL2> <S_LVALUES> lvalues() <S_ENDFILL>)#T_Fill2_Lvalues
}


/* Simple expressions */

void s_expression() #void : {}
{
  a_expression()
}

void a_expressions() #void : {}
{
  a_expression()#T_Expressions
}

void a_expression() #void : {}
{
  term() (
    <S_PLUS> term()#T_Plus(2)
|   <S_MINUS> term()#T_Minus(2)
```

```
/* Set Op */
|   <S_CONCAT> term()#T_Concat(2)
|   <S_UNION> term()#T_Union(2)
|   <S_INTERSECT> term()#T_Intersection(2)
)*
}

void term() #void : {}
{
  factor() (
    <S_TIMES> factor()#T_Times(2)
|   <S_SLASH> factor()#T_Divide(2)
|   <S_MOD> factor()#T_Mod(2)
|   <S_DIV> factor()#T_Div(2)
/* Set Op */
|   <S_BACKSLASH> factor()#T_Set_Diff(2)
  )*
}

void factor() #void : {}
{
  (true_factor()
| (<S_MINUS> factor())#T_Negate)

  (

/* Get expressions */
  <S_CARET> a_expression()#T_Get_n(2)
| <S_CARET><S_CARET> s_expression()#T_Get(2)

/*Array ref*/
| <S_LBRACKET> a_expressions() (
    <S_RBRACKET>#T_Aref(2)
|   <S_DOTDOT> (
      <S_RBRACKET>#T_Final_Seg(2)
|     a_expression() <S_RBRACKET>#T_Sub_Seg(3)
    )
|   <S_COMMA> a_expression() <S_RBRACKET>#T_Rel_Seg(3)
  )

/* Struct */
|   (T_Struct())

/* Gethash */
| <S_FULLSTOP> <S_LPAREN> expression() <S_RPAREN>#T_Gethash(2)
```

```
  )*
}

void true_factor() #void : {}
{
  exp_atom() (<S_EXPONENT> factor()#T_Exponent(2))*
}

void exp_atom() #void : {}
{
    <S_LPAREN> condition() <S_RPAREN>
|   T_Number()
|   a_prefix_op()

/* Funct Call */
| (T_IdentifierName() <S_LPAREN> ((expression() (<S_COMMA> expression())*)*)#
    T_Expressions <S_RPAREN>)#T_Funct_Call
| (T_AtName() ((<S_LPAREN> (expression() (<S_COMMA> expression())*)* <S_RPAREN
    >)*)#T_Expressions)#T_MW_Funct_Call
| (<S_PLINK_XF> T_IdentifierName() <S_LPAREN> ((expression() (<S_COMMA>
    expression())*)*)#T_Expressions <S_RPAREN>)#T_X_Funct_Call

/* Place Expressions */
| <S_EXPN_PLACE>#T_Expn_Place
| <S_VAR_PLACE>#T_Var_Place

/* Pattern Expressions */
| T_Expn_Pat_One()
| T_Expn_Pat_Many()
| T_Expn_Pat_Any()

| T_Variable()

| if_expression()

/* Set constructs */
| T_String()
| T_Set()
| T_Sequence()
| numb_type()
| s_prefix_op()
}

void a_prefix_op() #void : {}
{
```

```
  (<S_ABS> <S_LPAREN> a_expression() <S_RPAREN>)#T_Abs
| (<S_FRAC> <S_LPAREN> a_expression() <S_RPAREN>)#T_Frac
| (<S_INT> <S_LPAREN> a_expression() <S_RPAREN>)#T_Int
| (<S_SGN> <S_LPAREN> a_expression() <S_RPAREN>)#T_Sgn
| (<S_ARRAY> <S_LPAREN> a_expression() <S_COMMA> a_expression() <S_RPAREN>)#
    T_Array
| (<S_MAX> <S_LPAREN> a_expression() <S_COMMA> a_expression() <S_RPAREN>)#T_Max
| (<S_MIN> <S_LPAREN> a_expression() <S_COMMA> a_expression() <S_RPAREN>)#T_Min
| (<S_LENGTH> <S_LPAREN> s_expression() <S_RPAREN>)#T_Length
| (<S_REVERSE> <S_LPAREN> s_expression() <S_RPAREN>)#T_Reverse
| (<S_REDUCE> <S_LPAREN> T_Name() <S_COMMA> s_expression() <S_RPAREN>)#T_Reduce
| (<S_HEAD> <S_LPAREN> s_expression() <S_RPAREN>)#T_Head
| (<S_LAST> <S_LPAREN> s_expression() <S_RPAREN>)#T_Last
}


/* Set/string/sequence expressions */

void T_Set() #void : {}
{
  (<S_LBRACE> expression() <S_VBAR> condition() <S_RBRACE>)#T_Set
}

void T_Sequence() #void : {}
{
  (<S_LANGLE> ((expression() (<S_COMMA> expression())*)*)#T_Expressions <
      S_RANGLE>)#T_Sequence
}

void numb_type() #void : {}
{
  <S_RATS>#T_0
| <S_REALS>#T_0
| <S_NATS>#T_0
| <S_INTS>#T_0
}

void s_prefix_op() #void : {}
{
  (<S_MAP> <S_LPAREN> T_Name() <S_COMMA> s_expression() <S_RPAREN>)#T_Map
| (<S_POWERSET> <S_LPAREN> s_expression() <S_RPAREN>)#T_Powerset
| (<S_TAIL> <S_LPAREN> s_expression() <S_RPAREN>)#T_Tail
| (<S_BUTLAST> <S_LPAREN> s_expression() <S_RPAREN>)#T_Butlast
| (<S_SLENGTH> <S_LPAREN> s_expression() <S_RPAREN>)#T_Slength
| (<S_SUBSTR> <S_LPAREN> expressions() <S_RPAREN>)#T_Substr
```

```
| (<S_INDEX> <S_LPAREN> expressions() <S_RPAREN>)#T_Index
| (<S_REDUCE> <S_LPAREN> T_Name() <S_COMMA> s_expression() <S_RPAREN>)#T_Reduce
| (<S_HEAD> <S_LPAREN> s_expression() <S_RPAREN>)#T_Head
| (<S_LAST> <S_LPAREN> s_expression() <S_RPAREN>)#T_Last
}


/*
 * Terminals with values
 * ================
 */

void T_Cond_Pat_One() #T_Cond_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Cond_Pat_Many() #T_Cond_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Cond_Pat_Any() #T_Cond_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Assign_Pat_One() #T_Assign_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Assign_Pat_Many() #T_Assign_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Assign_Pat_Any() #T_Assign_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Expn_Pat_One() #T_Expn_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Expn_Pat_Many() #T_Expn_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Expn_Pat_Any() #T_Expn_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Lvalue_Pat_One() #T_Lvalue_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Lvalue_Pat_Many() #T_Lvalue_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}
```

```
void T_Lvalue_Pat_Any() #T_Lvalue_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Stat_Pat_One() #T_Stat_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Stat_Pat_Many() #T_Stat_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Stat_Pat_Any() #T_Stat_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Action_Pat_One() #T_Action_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Action_Pat_Many() #T_Action_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Action_Pat_Any() #T_Action_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Defn_Pat_One() #T_Defn_Pat_One :
{Token t;}{<S_PAT_ONE> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Defn_Pat_Many() #T_Defn_Pat_Many :
{Token t;}{<S_PAT_MANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Defn_Pat_Any() #T_Defn_Pat_Any :
{Token t;}{<S_PAT_ANY> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_String() #T_String :
{Token t;}{t=<S_STRING>{jjtThis.setValue(t.image.substring(1,t.image.length()
    -1));}}

void T_Number() #T_Number :
{Token t;}{t=<S_NUMBER>{jjtThis.setValue(t.image);}}

void T_Variable() #T_Variable :
{Token t;}{t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Name() #T_Name :
{Token t;}{t=<S_STRING>{jjtThis.setValue(t.image.substring(1,t.image.length()
    -1));}}

void T_IdentifierName() #T_Name :
```

```
{Token t;}{t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_AtName() #T_Name :
{Token t;}{<S_AT> (t=<S_IDENTIFIER> | t=<S_AMBIGOUS_IDENTIFIER>) {if (t.image.
    startsWith("@")) jjtThis.setValue(t.image);else jjtThis.setValue("@"+t.
    image);}}

void T_AtPatOneName() #T_Name :
{Token t;}{<S_AT_PAT_ONE> (t=<S_IDENTIFIER> | t=<S_AMBIGOUS_IDENTIFIER>) {if (t
    .image.startsWith("@")) jjtThis.setValue(t.image);else jjtThis.setValue("@
    "+t.image);}}

void T_Var_Lvalue() #T_Var_Lvalue :
{Token t;}{t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Exit() #T_Exit :
{Token t;}{<S_EXIT> <S_LPAREN> t=<S_NUMBER> <S_RPAREN>{jjtThis.setValue(t.image
    );}}

void T_Comment() #T_Comment :
{Token t;}{<S_COMMENT> t=<S_STRING>{jjtThis.setValue(t.image.substring(1,t.
    image.length()-1));}}

void T_Call() #T_Call :
{Token t;}{<S_CALL> t=<S_IDENTIFIER>{jjtThis.setValue(t.image);}}

void T_Struct_Lvalue() #T_Struct_Lvalue(2) :
{Token t;}{<S_FULLSTOP> T_IdentifierName()
{Node n = jjtThis.jjtGetChild(1); jjtThis.jjtSetChild(1,jjtThis.jjtGetChild(0))
    ; jjtThis.jjtSetChild(0,n);}}

void T_Struct() #T_Struct(2) :
{Token t;}{<S_FULLSTOP> T_IdentifierName()
{Node n = jjtThis.jjtGetChild(1);jjtThis.jjtSetChild(1,jjtThis.jjtGetChild(0));
    jjtThis.jjtSetChild(0,n);}}

/*
 * Tree manipulation routines
 * ====================
 */

/*
 *  This routine creates an pseudo else with a skip
 */
void pseudo_else() #T_Guarded :
```

```
{
}
{
  {
    ASTT_Statements statements = new ASTT_Statements(WSLParserTreeConstants.
        JJTT_STATEMENTS);
    statements.jjtAddChild(new ASTT_Skip(WSLParserTreeConstants.JJTT_SKIP),0);
    jjtThis.jjtAddChild(new ASTT_True(WSLParserTreeConstants.JJTT_TRUE),0);
    jjtThis.jjtAddChild(statements,1);
  }
}
```

Listing B.1: Parser Definition for Untyped WSL

Appendix C

FermaT Transformation Catalogues

C.1 Transformation Catalogue of fermat3 (Open Source Version)

Name	Description	Keywords
Abort Processing	Simplify statement sequences containing an ABORT.	Simplify
Absorb Left	This transformation will absorb into the selected statement the one that precedes it.	Join
Absorb Right	This transformation will absorb into the selected statement the one that follows it.	Join
Actions to Where	Convert an Action System to a Where clause	Rewrite,Simplify
Add Assertion	This transformation will add an assertion after the current item, if some suitable information can be ascertained.	Insert
Add Left	This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.	Join
Align Nested Statements	This transformation takes a guarded clause whose first statement is a If and integrates it with the outer condition by absorbing the other guarded statements into the inner If, and then modifying its conditions appropriately. This is the converse of Partially Join Cases.	Rewrite,L_to_R,R_to_L
Apply To Right	This transformation will apply the current program item to the one to its immediate right. For example, if the current item is an assertion and the next item is an IF statement, then the transformation will attempt to simplify the conditions using the assertions.	UseApply
Collapse Action System	Collapse action system will use simplifications and substitution to transform an action system into a sequence of statements, possibly inside a DO loop.	Rewrite
Collapse All Action Systems	Collapse All Action Systems will attempt to collapse the action systems within a program which is a WHERE structure.	Rewrite
Combine Where Structures	Combine Where Structures will combine two nested WHERE structures into one structure which will contain the definitions from each of the original WHERE structures.	Rewrite
Constant Propagation	Constant Propagation finds assignments of constants	Simplify
D Do To Floop	Convert any D_Do loop to a DO...OD loop	Rewrite
Delete All Assertions	This transformation will delete all the ASSERT statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the re-instatement of ASSERT or SKIP statements.	Simplify,L_to_R,R_to_L

Delete All Comments	This transformation will delete all the COMMENT statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the insertion of SKIP statements.	Simplify,L_to_R,R_to_L
Delete All Redundant	Delete All Redundant searches for redundant statements	Delete
Delete All Skips	This transformation will delete all the SKIP statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the re-instatement of SKIP statements.	Simplify,L_to_R,R_to_L
Delete Item	This transformation will delete a program item that is redundant or unreachable	Delete
Delete Redundant Registers	Delete Redundant Registers uses dataflow analysis	Delete
Delete Redundant Statement	Delete Redundant Statement checks whether	Delete
Delete Unreachable Code	Delete Unreachable Code will remove unreachable statements in the selected object. It will also remove unreachable cases in an IF statement, e.g those which follow a TRUE guard	Simplify
Delete What Follows	Delete What Follows will delete the code which follows the selected item if it can never be executed	UseApply
Double to Single Loop	Double to Single Loop will convert a double nested loop to a single loop, if this can be done without significantly increasing the size of the program.	Rewrite
Else If To Elsif	This transformation will replace an Else clause which contains an If statement with an Elsif clause. The transformation can be selected with either the outer If statement, or the Else clause selected.	Rewrite
Elsif To Else If	This transformation will replace an Elsif clause in an If statement with an Else clause which itself contains an If statement.	Rewrite
Expand And Separate All	Expand And Separate All will attempt to apply the transformation Expand and Separate to the first statement in each action in an action system. This will be useful for dealing with the skip_flag in WSL derived from Herma	Simplify
Expand And Separate	Expand And Separate will expand the selected IF statement to include all the following statements, then separate all possible statements from the resulting IF. This is probably only useful if the IF includes a CALL, EXIT etc. which is duplicated in the following statements, otherwise it will probably achieve nothing.	Reorder
Expand Call	Expand_Call will replace a call to an action, procedure or function with the corresponding definition.	Rewrite

Expand Forward	Expand_Forward will copy the following statement into the end of each branch of the selected IF or D_IF statement. It differs from Absorb Right in that the statement is only absorbed into the top level of the selected IF	Join
Find Terminals	Find and mark the terminal statements in the selected statement.	Rewrite
Fix Dispatch	This transformation will search for simple	Complex,Rewrite
Floop To While	Convert a suitable DO...OD loop to a While loop	Rewrite
For To While	Convert any FOR loop to a VAR plus WHILE loop	Rewrite
Force Double - Single Loop	Force Double - Single Loop will convert a double nested loop to a single loop, regardless of any increase in program size which this causes	Rewrite
Fully Absorb Right	This transformation will absorb into the selected statement all the statements that follow it.	Join
Fully Expand Forward	Apply Expand Forward as often as possible	Join
Globals To Pars	Convert global variables in procs to extra VAR parameters.	Rewrite
Insert Assertion(s)	This transformation will add an assertion inside the current item, if some suitable information can be ascertained.	Insert
Join All Cases	This transformation will join any guards in an If statement which contain the same sequence of statements (thus reducing their number) by changing the conditions of all the guards as appropriate.	Rewrite,Join
Make Procedure	Make Procedure will make a procedure from the body of an action or from a list of statements.	Rewrite
Merge Calls in Action	Merge Calls in Action will attempt to merge calls which call the same action, in the selected action	Simplify
Merge Calls	Use absorption to reduce the number of calls in an action system.	Simplify
Merge Cond Right	Merge a binary Cond with a subsequent Cond which uses the same	Simplify
Merge Left	This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.	Join
Merge Right	This transformation will merge the selected statement into the statement that precedes it.	Join
Meta Trans	Convert a FOREACH with a long sequence of IFMATCH commands to a more efficient form	Simplify
Move Comment Left	Moves the selected Comment Left.	Move
Move Comment Right	Moves the selected Comment Right.	Move

Move Comments	Move Comments will move any comments which appear at the end of actions within an action system and which follow a call. The comments will be moved in front of the call. This will help tidy up the output of the Herma translator.	Rewrite
Move To Left	This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.	Move
Move To Right	This transformation will move the selected item to the right so that it is exchanged with the item that follows it.	Move
Partially Join Cases	This transformation will join any guards in an If statement which contain almost the same sequence of statements (thus reducing their number) by introducing a nested If and changing the conditions of all the guards as appropriate.	Rewrite,Join
Prog To Spec	Convert given program to an equivalent specification statement.	Abstraction
Prune Dispatch	Simplify the dispatch action by removing references	Simplify
Push Pop	Look for a statement sequence with a PUSH of a var followed by a POP	Rewrite
Remove Recursion in Action	Remove Recursion in Action will replace the body of a recursive action if possible by an equivalent loop structure.	Rewrite
Reduce Loop	Automatically make the body of a DO...OD reducible (by introducing new procedures as necessary) and either remove the loop (if it is a dummy loop) or convert the loop to a WHILE loop (if the loop is a proper sequence).	Simplify
Reduce Multiple Loops	This transformation will reduce the number of multiply nested loops to a minimum.	Simplify
Refine Spec	Refine a specification statement into something closer to an implementation	Refinement
Remove All Redundant Vars	Remove All Redundant Vars applys Remove Redundant Vars	Delete
Remove Galileo Comments	Removes Galileo comments without a sequence number (SSL or SSE).	Delete
Remove Dummy Loop	Remove Dummy Loop will remove a DO loop which is redundant	Simplify
Remove Redundant Vars	Remove Redundant Vars takes out as many local variables	Delete
Rename Defns	Rename PROC definitions to avoid name clashes.	Rewrite
Rename Local Vars	Remove all local VAR statements by renaming the variables.	Rewrite
Rename Proc	Rename a PROC to given new name	Rewrite

Replace Accs With Value	This transformation will apply Replace With Value	Rewrite
Replace With Value	This transformation will replace a variable	Rewrite
Restore Local Vars	Try to restore the local var clauses	Rewrite
Reverse Order	This transformation will reverse the order of most two-component items; in particular expressions, conditions and Ifs which have two branches.	Reorder
– Separate –	– Separate – will take code out to the right and the left of the selected structure.	Reorder
– Separate	– Separate will take code out to the left of the selected structure. As much code As possible will be taken out; if all the statements are taken out then the original containing structure will be removed	Reorder
Separate –	Separate – will take code out to the right of the selected structure.	Reorder
Simple Slice	Perform Simple Slicing on a subset of WSL. Enter the list of variables to slice on as the data parameter.	Simplify
Simplify Action System	Simplify action system will attempt to remove actions and calls from an action system by successively applying simplifying transformations. As many of the actions as possible will be eliminated without making the program significantly larger.	Simplify
Simplify	This transformation will simplify any component as fully as possible.	Simplify,L_to_R,R_to_L
Simplify If	Simplify If will remove false cases from an IF statement, and any cases whose conditions imply earlier conditions. Any repeated statements which can be taken outside the if will be, and the conditions will be simplified if possible.	Simplify
Simplify Item	This transformation will simplify an item, but not recursively simplify the components inside it. In particular, the transformation will simplify expressions, conditions and degenerate conditional, local variable and loop statements.	Simplify,L_to_R,R_to_L
Static Single Assignment	Convert WSL code to Static Single Assignment form	Rewrite
Substitute and Delete	Substitute and Delete will replace all calls to an action, procedure or function with the corresponding definition, and delete the definition	Rewrite
Substitute and Delete List	Substitute and Delete List will replace all calls to any action	Rewrite
Syntactic Slice	Perform Syntactic Slicing using SSA and control dependencies. Enter the list of variables to slice on as the data parameter.	Simplify
Take Out Left	This transformation will take the selected item out of the enclosing structure towards the left.	Move

Take Out Of Loop	This transformation will take the selected item out of an appropriate enclosing loop towards the right.	Move
Take Out Right	This transformation will take the selected item out of the enclosing structure towards the right.	Move
Unfold Proc Call	Unfold the selected procedure call, replacing it with a copy of the procedure body.	Rewrite
Unfold Proc Calls	Unfold Proc Calls searches for procedures which are only called once, unfolds the call and removes the procedure.	Simplify
Use Assertion	Use the currently selected assertion to simplify code.	Simplify
Var Pars To Val Pars	Add all VAR pars as extra value pars where needed.	Rewrite
While To Abort	This transformation replaces a non-terminating while loop with a conditional abort	Simplify,L_to_R,R_to_L
While To Floop	Convert any WHILE loop to a DO...OD loop	Rewrite

C.2 Transformation Catalogue of fermat2 (Commercial Version)

Abort Processing	Simplify statement sequences containing an ABORT.	Simplify
Absorb Left	This transformation will absorb into the selected statement the one that precedes it.	Join
Absorb Right	This transformation will absorb into the selected statement the one that follows it.	Join
Actions to Procs	Search for actions which call one other action and make them into procs.	Rewrite,Simplify
Actions to Where	Convert an Action System to a Where clause	Rewrite,Simplify
Add Assertion	This transformation will add an assertion after the current item, if some suitable information can be ascertained.	Insert
Add Left	This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.	Join
Add Loop To Action	If an action is only called by one other action,	Simplify
Align Nested Statements	This transformation will apply the current program item to the one to its immediate right. For example, if the current item is an assertion and the next item is an IF statement, then the transformation will attempt to simplify the conditions using the assertions.	Rewrite
Apply To Right	This transformation will apply the current program item to the one to its immediate right. For example, if the current item is an assertion and the next item is an IF statement, then the transformation will attempt to simplify the conditions using the assertions.	Use/Apply
Collapse Action System	Collapse action system will use simplifications and substitution to transform an action system into a sequence of statements, possibly inside a DO loop.	Rewrite
Collapse All Action Systems	Collapse All Action Systems will attempt to collapse the action systems within a program which is a WHERE structure.	Rewrite
Combine Where Structures	Combine Where Structures will combine two nested WHERE structures into one structure which will contain the definitions from each of the original WHERE structures.	Rewrite
Constant Propagation	Constant Propagation finds assignments of constants	Simplify
D Do To Floop	Convert any D_Do loop to a DO...OD loop	Rewrite
Data Translation A	Data Translation translates array references	Rewrite
Date Find	Find direct and indirect references to dates for Y2000 analysis.	Simplify
Decompile	This transformation will apply some basic	Simplify

Delete All Assertions	This transformation will delete all the ASSERT statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the re-instatement of ASSERT or SKIP statements.	Simplify
Delete All Comments	This transformation will delete all the COMMENT statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the insertion of SKIP statements.	Simplify
Delete All Redundant	Delete All Redundant searches for redundant statements	Delete
Delete All Skips	This transformation will delete all the SKIP statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the re-instatement of SKIP statements.	Simplify
Delete Item	This transformation will delete a program item that is redundant or unreachable	Delete
Delete Redundant Registers	Delete Redundant Registers uses dataflow analysis	Delete
Delete Redundant Statement	Delete Redundant Statement checks whether	Delete
Delete Savearea	Delete code from assembler translations that looks	Simplify
Delete Unreachable Code	Delete Unreachable Code will remove unreachable statements in the selected object. It will also remove unreachable cases in an IF statement, e.g those which follow a TRUE guard	Simplify
Delete What Follows	Delete What Follows will delete the code which follows the selected item if it can never be executed	Use/Apply
Double to Single Loop	Double to Single Loop will convert a double nested loop to a single loop, if this can be done without significantly increasing the size of the program.	Rewrite
Else If To Elsif	This transformation will replace an Else clause which contains an If statement with an Elsif clause. The transformation can be selected with either the outer If statement, or the Else clause selected.	Rewrite
Elsif To Else If	This transformation will replace an Elsif clause in an If statement with an Else clause which itself contains an If statement.	Rewrite
Error Processing	Replace ERROR proc bodies by ABORTs.	Simplify
Expand And Separate All	Expand And Separate All will attempt to apply the transformation Expand and Separate to the first statement in each action in an action system. This will be useful for dealing with the skip_flag in WSL derived from Herma	Simplify

Expand And Separate	Expand And Separate will expand the selected IF statement to include all the following statements, then separate all possible statements from the resulting IF. This is probably only useful if the IF includes a CALL, EXIT etc. which is duplicated in the following statements, otherwise it will probably achieve nothing.	Reorder
Expand Call	Expand_Call will replace a call to an action, procedure or function with the corresponding definition.	Rewrite
Expand Forward	Expand_Forward will copy the following statement into the end of each branch of the selected IF or D_IF statement. It differs from Absorb Right in that the statement is only absorbed into the top level of the selected IF	Join
F2K 1	Apply the initial sequence of F2K transformations	Hidden
F2K 2	Apply the final sequence of F2K transformations	Hidden
Find Dead Code	Find dead code in the action system	Rewrite
Find Edits	Searches for !P ed(...) calls and then searches back	Simplify
Find Entry Points	Find possible entry points in the action system	Rewrite
Find Parameters	Find_Parameters looks for !P calls	Rewrite
Find Terminals	Find and mark the terminal statements in the selected statement.	Rewrite
Fix Assembler (370)	This transformation will apply some basic	Simplify
Fix Assembler (Fast)	Faster version of Fix_Assembler – for Y2000	Simplify
Fix Assembler (Slice)	Version of Fix_Assembler for slicing	Simplify
Fix Assembler (x86)	This transformation will apply some basic	Simplify
Fix Calls	Fix_Calls looks for !P calls	Rewrite
Fix Decimal	Fix Decimal converts simple p_lit calls to	Rewrite
Fix Dispatch	This transformation will search for simple	Complex,Rewrite
Fix Endian	Fix_Endian looks for integers which have part of their	Rewrite
Fix For Slicing	Apply transformations to fix a WSL file before generating a df file for assembler slicing	Rewrite
Fix Init	Delete assignments from __rX_init__ and push_regs / pop_regs	Simplify
Fix Parameters	Fix_Parameters looks for !P calls	Rewrite
Flag Removal	Attempt to remove references to flag variables	Simplify
Floop To While	Convert a suitable DO...OD loop to a While loop	Rewrite
For To While	Convert any FOR loop to a VAR plus WHILE loop	Rewrite
Force Double - Single Loop	Force Double - Single Loop will convert a double nested loop to a single loop, regardless of any increase in program size which this causes	Rewrite
Fully Absorb Right	This transformation will absorb into the selected statement all the statements that follow it.	Join
Fully Expand Forward	Apply Expand Forward as often as possible	Join
Globals To Pars	Convert global variables in procs to extra VAR parameters.	Rewrite

If To Case	Convert nested IF statements which test a variable against constant	Simplify
Insert Assertion(s)	This transformation will add an assertion inside the current item, if some suitable information can be ascertained.	Insert
Join All Cases	This transformation will join any guards in an If statement which contain the same sequence of statements (thus reducing their number) by changing the conditions of all the guards as appropriate.	Rewrite,Join
Loop To Move	Convert a suitable DO...OD or WHILE loop to assignments.	Rewrite
Make Procedure	Make Procedure will make a procedure from the body of an action or from a list of statements.	Rewrite
Make Reducible	Use absorption if necessary to make the selected item	Rewrite
Merge Calls in Action	Merge Calls in Action will attempt to merge calls which call the same action, in the selected action	Simplify
Merge Calls	Use absorption to reduce the number of calls in an action system.	Simplify
Merge Cond Right	Merge a binary Cond with a subsequent Cond which uses the same	Simplify
Merge Left	This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.	Join
Merge Right	This transformation will merge the selected statement into the statement that precedes it.	Join
Meta Trans	Convert a FOREACH with a long sequence of IFMATCH commands to a more efficient form	Simplify
Move Comment Left	Moves the selected Comment Left.	Move
Move Comment Right	Moves the selected Comment Right.	Move
Move Comments	Move Comments will move any comments which appear at the end of actions within an action system and which follow a call. The comments will be moved in front of the call. This will help tidy up the output of the Herma translator.	Rewrite
Move To Left	This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.	Move
Move To Right	This transformation will move the selected item to the right so that it is exchanged with the item that follows it.	Move

Partially Join Cases	This transformation will join any guards in an If statement which contain almost the same sequence of statements (thus reducing their number) by introducing a nested If and changing the conditions of all the guards as appropriate.	Rewrite,Join
Prog To Spec	Convert given program to an equivalent specification statement.	Abstraction
Prune Dispatch	Simplify the dispatch action by removing references	Simplify
Push Pop	Look for a statement sequence with a PUSH of a var followed by a POP	Rewrite
Raise Abstraction	Raise the level of abstraction of a WSL program.	Simplify
Remove Recursion in Action	Remove Recursion in Action will replace the body of a recursive action if possible by an equivalent loop structure.	Rewrite
Reduce Loop	Automatically make the body of a DO...OD reducible (by introducing new procedures as necessary) and either remove the loop (if it is a dummy loop) or convert the loop to a WHILE loop (if the loop is a proper sequence).	Simplify
Reduce Multiple Loops	This transformation will reduce the number of multiply nested loops to a minimum.	Simplify
Reduce Nots	Reduce the number of negations (T_Not and T_Not_Equal types) in the conditions in an IF statement by switching the order of the guards.	Simplify,Rewrite
Refine Spec	Refine a specification statement into something closer to an implementation	Refinement
Remove All Redundant Vars	Remove All Redundant Vars applys Remove Redundant Vars	Delete
Remove Galileo Comments	Removes Galileo comments without a sequence number (SSL or SSE).	Delete
Remove Dummy Loop	Remove Dummy Loop will remove a DO loop which is redundant	Simplify
Remove Redundant Vars	Remove Redundant Vars takes out as many local variables	Delete
Rename Defns	Rename PROC definitions to avoid name clashes.	Rewrite
Rename Local Vars	Remove all local VAR statements by renaming the variables.	Rewrite
Rename Proc	Rename a PROC to given new name	Rewrite
Replace Accs With Value	This transformation will apply Replace With Value	Rewrite
Replace With Value	This transformation will replace a variable	Rewrite
Replace With Variable	This transformation will search for a variable	Rewrite
Restore Local Vars	Try to restore the local var clauses	Rewrite
Reverse Order	This transformation will reverse the order of most two-component items; in particular expressions, conditions and Ifs which have two branches.	Reorder

Roll Loop	Roll the first step of a WHILE loop.	Rewrite
Semantic Slice	Perform Semantic Slicing on a subset of WSL. Enter the list of variables to slice on as the data parameter.	Simplify
– Separate –	– Separate – will take code out to the right and the left of the selected structure.	Reorder
Separate Exit Code	Separate Exit Code will take exit code (code which must lead to termination of the loop) out of the loop, using a flag if necessary that indicates which exit from the loop was taken.	Reorder
– Separate	– Separate will take code out to the left of the selected structure. As much code as possible will be taken out; if all the statements are taken out then the original containing structure will be removed	Reorder
Separate –	Separate – will take code out to the right of the selected structure.	Reorder
Simple Slice	Perform Simple Slicing on a subset of WSL. Enter the list of variables to slice on as the data parameter.	Simplify
Simplify Action System	Simplify action system will attempt to remove actions and calls from an action system by successively applying simplifying transformations. As many of the actions as possible will be eliminated without making the program significantly larger.	Simplify
Simplify	This transformation will simplify any component as fully as possible.	Simplify
Simplify If	Simplify If will remove false cases from an IF statement, and any cases whose conditions imply earlier conditions. Any repeated statements which can be taken outside the if will be, and the conditions will be simplified if possible.	Simplify
Simplify Item	This transformation will simplify an item, but not recursively simplify the components inside it. In particular, the transformation will simplify expressions, conditions and degenerate conditional, local variable and loop statements.	Simplify
Sort If	Sort the branches of IF statements depending on	Rewrite
Sort Procs	Sort the order of procs in a WHERE so that:	Rewrite
Sort Tests	Sort tests so that all other tests come *before*	Rewrite
Static Single Assignment	Convert WSL code to Static Single Assignment form	Rewrite
Substitute and Delete	Substitute and Delete will replace all calls to an action, procedure or function with the corresponding definition, and delete the definition	Rewrite
Substitute and Delete List	Substitute and Delete List will replace all calls to any action	Rewrite

Syntactic Slice	Perform Syntactic Slicing using SSA and control dependencies. Enter the list of variables to slice on as the data parameter.	Simplify
Tail Calls	Find !P calls from an entry point to itself	Rewrite
Take Out Left	This transformation will take the selected item out of the enclosing structure towards the left.	Move
Take Out Of Loop	This transformation will take the selected item out of an appropriate enclosing loop towards the right.	Move
Take Out Right	This transformation will take the selected item out of the enclosing structure towards the right.	Move
Unfold Dynamic Calls	Look for small procedures which contain a !P call_via_ptr(rx) and unfold them everywhere	Rewrite
Unfold Proc Call	Unfold the selected procedure call, replacing it with a copy of the procedure body.	Rewrite
Unfold Proc Calls	Unfold Proc Calls searches for procedures which are only called once, unfolds the call and removes the procedure.	Simplify
Unroll Loop	Unroll the first step of a WHILE loop.	Rewrite
Use Assertion	Use the currently selected assertion to simplify code.	Simplify
Var Pars To Val Pars	Add all VAR pars as extra value pars where needed.	Rewrite
While To Abort	This transformation replaces a non-terminating while loop with a conditional abort	Simplify
While To Floop	Convert any WHILE loop to a DO...OD loop	Rewrite
While To Reduce	Replace a WHILE loop with an equivalent REDUCE or MAP	Simplify
x86 Fix	Fixes for code translated from x86 assembler.	Rewrite
X86 Proc	Convert an entry point with !P calls into a local PROC.	Rewrite

Appendix D

FermaT Maintenance Environment Tutorial

FermaT Maintenance Environment Tutorial

FermaT Maintenance Environment Tutorial

Matthias Ladkau (matthias@ladkau.de)

Abstract

This document gives a brief practical oriented introduction to the FermaT Maintenance Environment (FME). It covers the installation and usage of the program via practical examples.

Contents

1 Installation of the FermaT Maintenance Environment 3
1.1 Installation on Unix / Linux . . . 3
1.1.1 Requirements . . . 3
1.1.2 Install of Perl and gcc . . . 3
1.1.3 Installation of a JAVA environment . . . 3
1.1.4 Installation of Bit::Vector for perl . . . 4
1.1.5 Installation of Set-IntRange for perl . . . 4
1.1.6 Installation of FermaT Maintenance Environment . . . 5
1.2 Installation on Windows . . . 5
1.2.1 Requirements . . . 5
1.2.2 Install Windows Installer (for older versions of windows) 5
1.2.3 Installation of a JAVA environment . . . 6
1.2.4 Install active perl . . . 6
1.2.5 Install of Bit::Vector and Set-IntRange for perl . . . 6
1.2.6 Install gcc . . . 6
1.2.7 Installation of FermaT Maintenance Environment . . . 7

2 Applied Software Evolution With The FermaT toolset 8
2.1 FermaT Transformation System . . . 8
2.2 The Wide-Spectrum Language . . . 9
2.3 FermaT Maintenance Environment . . . 9

3 Interface 11
3.1 Project . . . 11
3.2 File . . . 11
3.3 Other functionalities . . . 12

4 Getting Started 13
4.1 First experience . . . 13

4.2 Transformation Example 15
4.3 Working with the console 17
4.4 Other functionalities of the FME 18

Bibliography **19**

Web Links **19**

1 Installation of the FermaT Maintenance Environment

This chapter gives a guidance through the installation process of the FermaT Maintenance Environment. The installation is explained for Unix/Linux and Windows operating systems.

1.1 Installation on Unix / Linux

1.1.1 Requirements

- Perl (version >= 5.6.1)
  http://www.cpan.org/
- Bit::Vector (A perl module for efficient sets of integers by Steffen Beyer)
  http://search.cpan.org/search?module=Bit::Vector
- Set::IntRange (Perl module based on Bit::Vector for sets of integers in a given range by Steffen Beyer)
  http://search.cpan.org/search?module=Set::IntRange
- gcc or a compatible C compiler
  http://www.gnu.org/software/gcc/gcc.html
- A working JAVA environment (version >= 6)
  http://java.sun.com/
- The "make" command
  http://java.sun.com/

1.1.2 Install of Perl and gcc

Perl and gcc are included in every current linux distribution. See the install instructions of your distributions if these components are not already installed in the standard installation.

1.1.3 Installation of a JAVA environment

On order for the FME to work correctly the Java environment of SUN should be used. The environment can directly optained from the SUN Microsystems as a cost-free download.

- Download the Java **JDK** for Unix/Linux from
  http://java.sun.com/javase/downloads/index.jsp

- Install it according to the provided instructions
- A quick solution is to download the self extracting version (without "-rpm" in the filename) and install it into your `/opt` directory. Create symbolic links in your /usr/bin directory to the `java` and `javac` executables in the `/bin` directory of the extracted java distribution.

1.1.4 Installation of Bit::Vector for perl

- Download Bit::Vector from CPAN
  http://search.cpan.org/search?module=Bit::Vector
- Unpack the archive:

```
tar zxvf Bit-Vector-6.4.tar.gz
```

- Change directory to the unpacked files:

```
cd Bit-Vector-6.4
```

- Make and install the binaries:

```
perl Makefile.PL
make
make install
```

1.1.5 Installation of Set-IntRange for perl

- Download Set::IntRange from CPAN Beyer)
  http://search.cpan.org/search?module=Set::IntRange
- Unpack the archive:

```
tar zxvf Set-IntRange-5.1.tar
```

- Change directory to the unpacked files:

```
cd Set-IntRange-5.1
```

- Make and install the binaries:

```
perl Makefile.PL
make
```

```
make install
```

1.1.6 Installation of FermaT Maintenance Environment

- Unpack the archive (note the directory path must not contain any space characters) :

  ```
  tar zxvf fme.tar.gz
  ```

- Change directory to the unpacked files:

  ```
  cd fme
  ```

- The program should run now by executing the `fme.sh` script

1.2 Installation on Windows

1.2.1 Requirements

- Active Perl (version >= 5.6)
  http://www.activestate.com/ActivePerl/
- Bit::Vector (A perl module for efficient sets of integers by Steffen Beyer)
  http://search.cpan.org/search?module=Bit::Vector
- Set::IntRange (Perl module based on Bit::Vector for sets of integers in a given range by Steffen Beyer)
  http://search.cpan.org/search?module=Set::IntRange
- MinGW package
  http://www.mingw.org/
- Windows Installer >2.0 (if using older versions of windows e.g. Win9x/WinME)
  http://downloads.activestate.com/contrib/Microsoft/MSI2.0/
- A working JAVA 6.0 environment
  http://java.sun.com/

1.2.2 Install Windows Installer (for older versions of windows)

- Install InstMsiA.exe when using Win9x/WinME or InstMsiW.exe for WinNT. The setup binaries can be found on the ActiveState website:

http://downloads.activestate.com/contrib/Microsoft/MSI2.0/

1.2.3 Installation of a JAVA environment

- Download the Java JDK for Windows from
  http://java.sun.com/javase/downloads/index.jsp
- Install it according to the provided instructions

1.2.4 Install active perl

- Download Active Perl (version >= 5.6)
  http://www.activestate.com/ActivePerl/
- Install ActivePerl with the installer.
  NOTE: The command "perl" should now work in a DOS box (Start->Run->cmd). To end perl press CTRL+C.

1.2.5 Install of Bit::Vector and Set-IntRange for perl

- Download and Install Bit::Vector and Set::IntRange through the ppm program from ActiveState. Open a DOS box and type at the prompt:

  ```
  ppm: install Bit-Vector
  ppm: install Set-IntRange
  ppm: quit
  ```

  If this doesn't work then the names might have changed. Try to search the "Set" and Bit" modules to get the right name:

  ```
  ppm: search Bit
  ```
  or
  ```
  ppm: search Set
  ```

1.2.6 Install gcc

- Install the MinGW (e.g. MinGW-6.0.2.exe)
- Extent the path variable for the gcc compiler

  In Windows XP and Vista:
  Start->Settings->Control Panel->System

  Pick Advanced tab

Click on "Environment Variables"

Search for "Path" variable in the system variables list

Click on edit

Append to the Variable value string the path of the gcc.exe

(e.g. when MinGW was installed to "`C:\Program Files\MinGW`" then append "`C:\Program Files\MinGW\bin`")

- The command "gcc" should now work in a DOS box

1.2.7 Installation of FermaT Maintenance Environment

- Unpack the archive (with WinZIP or WinRAR) into any directory (note the directory path must not contain any space characters)
- The program should run now by executing the `fme.bat` script

2 Applied Software Evolution With The FermaT toolset

2.1 FermaT Transformation System

The objective of the FermaT transformation system is to enable the migration of large, highly complex legacy systems from Assembler to higher-level language such as C or COBOL. The FermaT transformation system is built on the transformation theory that has the following objectives.

1. Improving the maintainability (in particular, flexibility and reliability, and
2. hence extending the lifetime) of existing mission-critical software systems;
3. Translating programs to modern programming languages;
4. Developing and maintaining safety-critical applications;
5. Extracting reusable components from current systems, deriving their specifications, and storing the specification, implementation, and development strategy in a repository for subsequent reuse;
6. Reverse engineering from existing systems to high-level specifications, followed by subsequent reengineering and evolutionary development;

Once migrated, these systems are substantially easier to maintain and can evolve faster to meet the changing needs of the business they support. Unlike simple line by line language migration technologies, the FermaT transformation's unique semantics preserving code transformations enable the original application to be automatically cleaned-up, simplified and restructured to its optimum state for migration to the chosen new language [4]. This ensures that only functional code is migrated to the new language, helping to ensure that the migrated code is significantly easier to maintain and adapt than the original.

Because of FermaT's use of a unique and formally defined high-level language, Wide Spectrum Language (WSL), and its specifically designed code transformations, the migration process can be automated [6]. The migration process of the FermaT transformation system can be divided into three basic steps:

1. Translation of the assembler to WSL;
2. Translate and restructure data declarations;

3. Apply semantics-preserving WSL to WSL transformations;

4. Translate the high-level WSL to the target language.

2.2 The Wide-Spectrum Language

The core of the FermaT transformation system is the WSL language. It is based on a wide spectrum language, using Morgan's specification statement [2] and Dijkstra's guarded commands [1]. The intention is to form a language which acts as an intermediate language when processing a legacy system [3].

WSL was designed for reengineering tasks and covers:

- Simple, regular and formally defined semantics
- Simple, clear and unambiguous syntax
- A wide range of transformations with simple, mechanically-checkable correctness conditions
- The ability to express low-level programs and high-level abstract specifications

The heart of the WSL language is a very small and mathematically tractable kernel language. This language supports already all necessary operations needed for a programming and specification language. In the context of this tiny kernel language it is relatively easy to prove the correctness of a transformation, but the language is not very expressive for programming. For that reason the language is extended into an expressive programming language by defining new constructs in terms of the kernel. This extension is carried out in a series of layers with each layer building on the previous language level (see [3] for details).

2.3 FermaT Maintenance Environment

The FME is written entirely in the Java language. The choice for the Java language was made because it is a very safe (strong typed) and flexible language with an extensive API and a vast amount of open-source libraries for almost every possible computer task. Primary the FME is a graphical interface to the FermaT transformation engine. It consists of a text editor which is able to express WSL together with an Abstract Syntax Tree (AST) viewer. A maintainer can navigate through the code via the source code or via the AST. The environment provides a console to the FermaT transformation engine which can be used to directly command the engine. The core

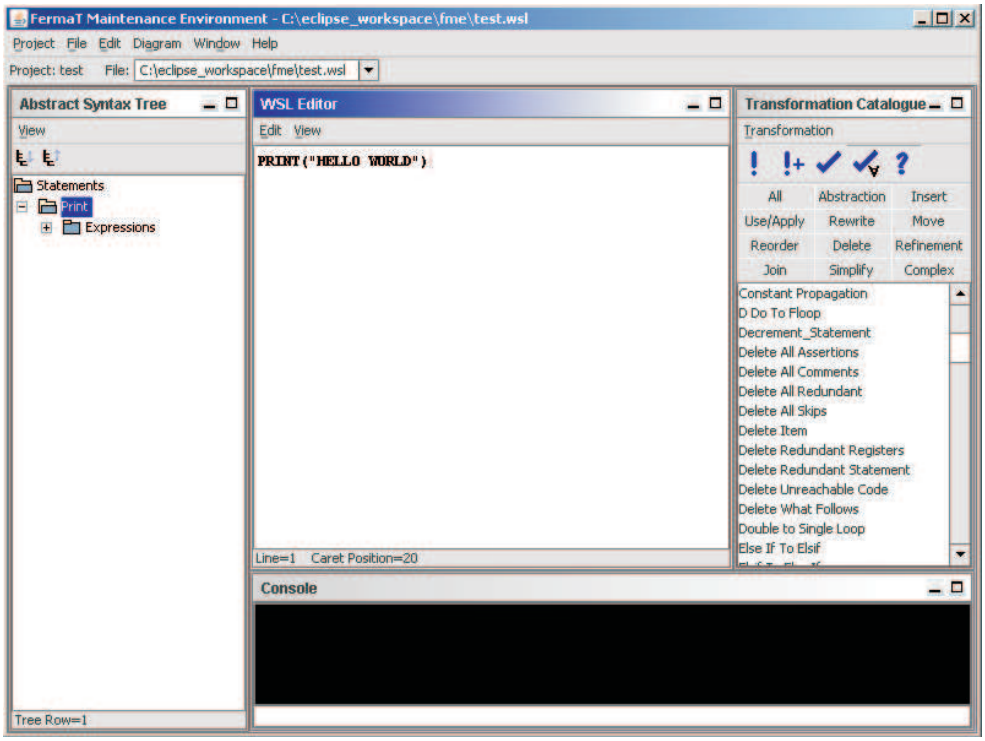

Figure 1: The FermaT Maintenance Environment

of the transformation engine is a set of mathematical proven program transformation to simplify the source code. The transformations can be selected from a transformation catalogue and performed on a chunk of WSL code. The transformations will either produce a semantically equivalent or refined version of the source program construct [3]. The communication between the FermaT Maintenance Environment and the FermaT transformation engine is provided through a communication pipe provided by the underlying operation system.

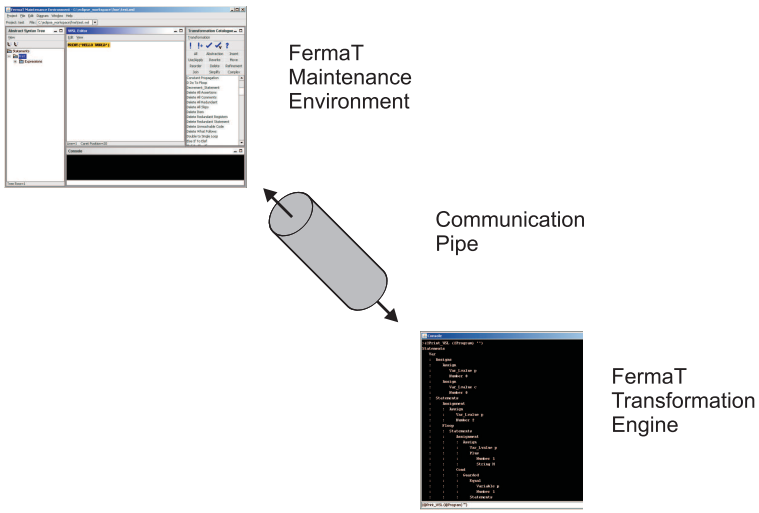

Figure 2: Communication between the Engine and the FME

3 Interface

Following sections give a brief explanation of the main functions of the FME.

3.1 Project

WSL files which are to be used with the FME must be organised in a project. An own directory should be reserved for the project and WSL files. A project can be created or modified with the Project Manager (Project→Show Project Manager). The Project Manager provides the following functions:

- New Project
  Create a new project.
- Open project
  Opens an existing project.
- Save project
  Saves the changes which have been done to the project.
- Close project
  Closes a project
- Open file from project
  Open a file from the project in the FME.
- Create empty file in project
  Creates an empty file and adds it to the project.
- Add file to project
  Add an existing file to the project. The file should be in the same directory as the project file.
- Remove file from project
  Removes a file from the project.
- Show Modification History
  Shows a graph with all applied transformations and saved intermediate WSL files.
- Delete History
  Resets the modification history record.

3.2 File

Every correct WSL file can be interpreted and executed by the underlying Scheme interpreter (File→Run WSL file). If a WSL file was loaded into

the FME and modified, it can safed as an intermediate version. In this case, the filename is extended with a version number e.g. <file>-1.wsl. Intermediate versions should be used throughout the whole transformation process. Every applieed transformation creates a new intermediate version. When the transformation process has finished the final WSL version can be saved wiht File as the original filename whereby all intermediate WSL files are deleted (File→Save code and delete intermediate version). If the intermediate versions should be kept it is also possible to save a WSL file to a different file with (File→Save code as WSL file).

3.3 Other functionalities

Apart from the main file handling functions the FME has many other functionalities:

- UNDO/REDO functions which can undo or redo any edit on a WSL file.
- The calls of Actions or Procedures within WSL files can be visualised in Diagrams.
- Find / Replace which can search or replace text patterns in WSL file.
- The editor has the standard copy, paste and cut abilities.
- Selected items in the Abstract Syntax Tree can be highlighted in the code and visa versa.

4 Getting Started

4.1 First experience

The followings are two examples written in WSL. First Example: Hello world in WSL

```
PRINT("Hello World!")
```

The output should look something like this when it is directly executed with the "wsl.pl" script of the engine:

```
Writing: C:\DOKUME~1\TheUser\LOKALE~1\Temp\t11436.scm
Starting Execution...

Hello World!

Execution time: 0
```

To get this result in the FME we create a new project and add an empty file. We then type the statement from above into the editor and save it by selecting "File"→"Save code as intermediate version". Note that the Abstract Syntax Tree in the FME is only updated after the file has been saved. When we now select "File"→"Run WSL File" the following window should appear:

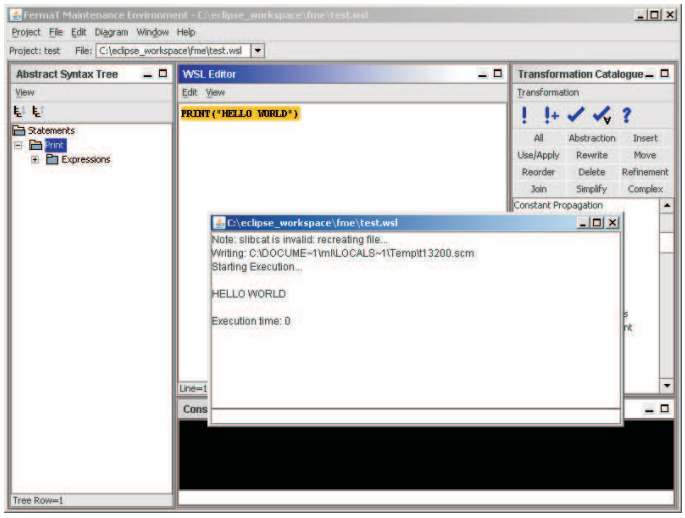

Figure 3: Execution of Hello World

The first line tells the interested reader that a temporary file "t11436.scm" has been written to the temporary directory of the current user (here under

the windows operating system). This is because the FermaT transformation engine converts a WSL program before execution into Scheme. The Scheme interpreter is than used to execute the program. This technique has a serious drawback: The runtime errors detected be Scheme will refer to lines in the Scheme programm. So the user isn't able to trace the error according to line numbers unless he knows exactly which WSL statement of his program is mapped to particular Scheme statement(s) in the executed program. For development language this would be a serious problem but as mentioned before the WSL language is intended to be an intermediate language for migration and analysing tasks rather than for software development tasks.

The second Example should demonstrate an interactive program - A simple guessing game in WSL:

```
VAR <num:=0,guess:=0>:
  num := @Random(100);
  PRINT("I have thought of a number between 1 and 100");
  DO PRINFLUSH("What is your guess? ");
    guess := @String_To_Num(@Read_Line(Standard_Input_Port));
    IF guess = num THEN PRINT("Correct!"); EXIT(1) FI;
    IF guess < num
      THEN PRINT("Too low.")
      ELSE PRINT("Too high.") FI OD;
  PRINT("Goodbye.")
ENDVAR
```

The output should look something like this:

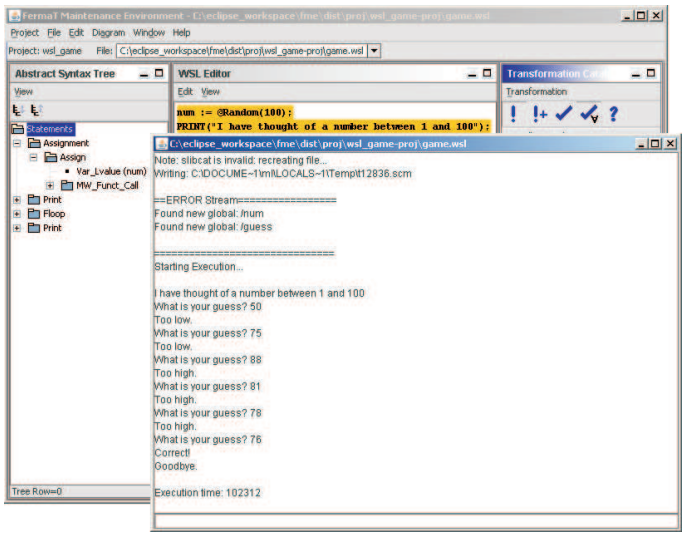

Figure 4: Execution of the guessing game

A more in-depth reference of all possible WSL statements can be found in [5].

4.2 Transformation Example

To demonstrate the transformation facility of FermaT we introduce a small program. The execution will just output "Hello World". The reader is encouraged to save the following chunk of code in a file within a project of his choice.

```
VAR < x := 0, y := 0 >:
DO DO IF x = 0 THEN PRINT("Hello World")
      ELSIF x > (2 + x) - 1
        THEN PRINT("Goodby cruel world")
        ELSE EXIT(2) FI;
      x := x + 1 OD OD ENDVAR
```

A click onto the first *DO* statement should highlight the whole loop:

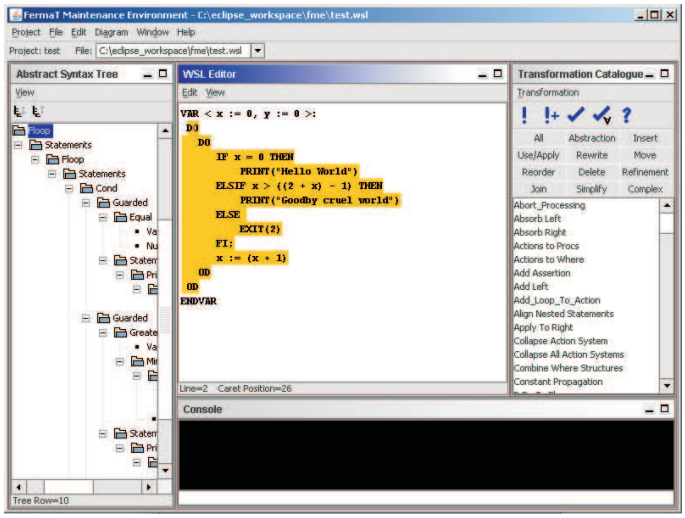

Figure 5: Transformation Example

We can test now for the available transformations on this node with a click on the "Test All Transformations" symbol in the transformation catalogue.

Figure 6: Transformation Catalogue Toolbar

Now some transformations should be highlighted in green. These transformations can be applied on the current selected program item (Of course some of them may have no effect).

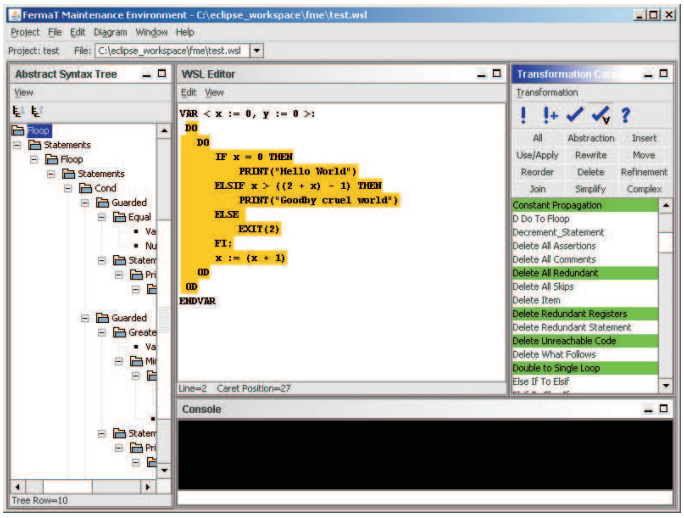

Figure 7: Transformation Example

Now we select the transformation "Double to Single Loop". A click on "Apply Transformation" should result in the following code:

```
VAR < x := 0, y := 0 >:
DO IF x = 0
     THEN PRINT("Hello World")
   ELSIF x > (2 + x) - 1
     THEN PRINT("Goodby cruel world")
     ELSE EXIT(1) FI;
   x := x + 1 OD ENDVAR
```

The double loop has been eliminated and the exit statement inside the loop have been decreased by one. After the transformation has been applied the attentive reader may have recognised that the filename contains now a "-0". Everytime the source code is modiefied and saved the FME will generate an new file (called "intermediate version"). This includes the case when a transformation is applied. If all modifications of the file have been finished the user may select the "Save (final) WSL File and all intermediate versions of the file will be deleted and the last version will replace now the original file. The benefit of this technique is that a maintainer can access in the process of migration all past intermediate versions of the processed program. If he did something wrong or applied the transformations in a wrong order he can easily go back to an older version.

If the file is not saved than all modifications can be undone/redone with the "undo" and "redo" options of the "Edit" menu entry.

Through the "Simplify If" Transformation the program can be simplified to[1]:

[1]The interested readers may do this on their own.

```
VAR < x := 0, y := 0 >:
DO IF x <> 0 THEN EXIT(1) FI;
   PRINT("Hello World");
   x := x + 1 OD ENDVAR
```

4.3 Working with the console

As mentioned before the FME is directly tied to the transformation engine. The engine itself can be accessed via the console. The current program and the current item[2] is changed when a node in the FME's Abstract Syntax Tree Window has been selected. To demonstrate this the reader may open a WSL program of his choice and select a node in the tree. This node is now the current item in the transformation engine. If now the command

```
(@Print_WSL (@I) "")
```

is entered in the Console the engine should output a subtree of the AST with the current item as the root.

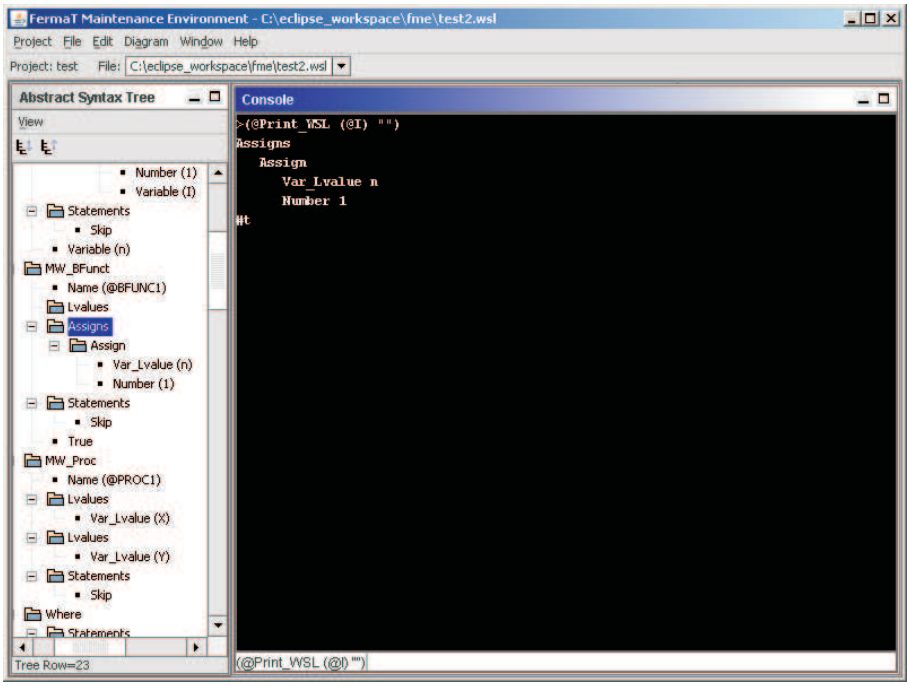

Figure 8: Console to the transformation engine

For all possible commands please see the WSL manual [5].

[2]See [5] for details on these concepts

4.4 Other functionalities of the FME

The FME includes some more functionalities for comfortable usage:

The graphical user interface of the FME consists of internal windows which may be minimised, maximised or closed. The "Window" menu entry gives some options to automatically align these windows.

The "ActionSystem CallGraph" is a very useful feature when analysing action systems.

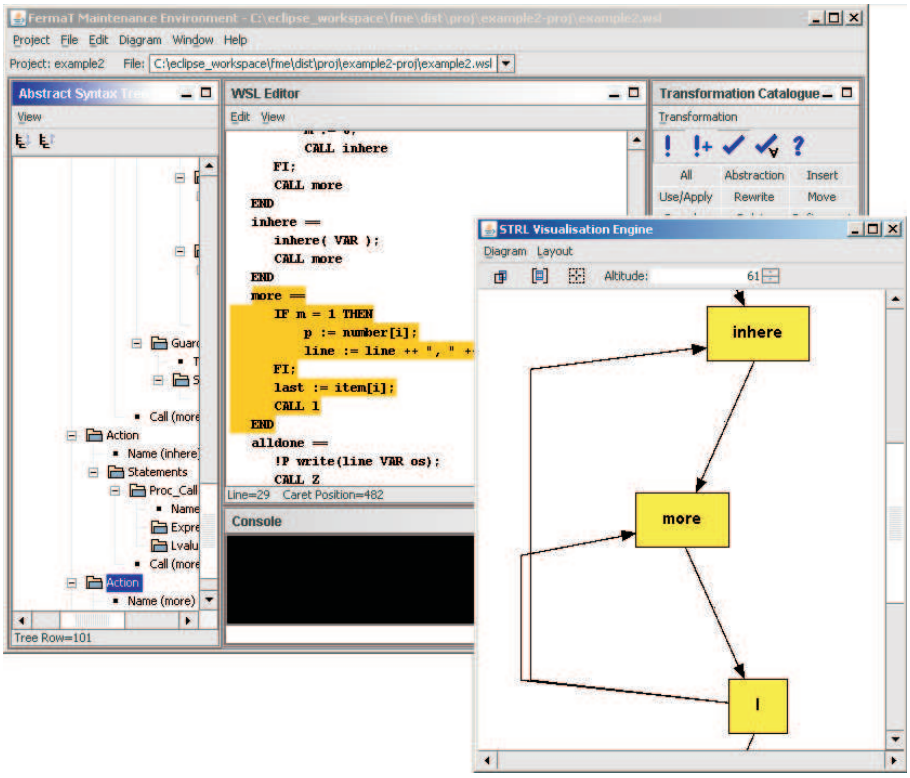

Figure 9: ActionSystem CallGraph

The dialog can be activated within the "Analyse" menu entry if an WSL Action System is present in the current program. It shows the calls in the action system in a call graph. The user can automatically zoom-in ,hide/collapse several nodes, print the graphic and export the graphic to a vector/bitmap file format.

References

[1] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[2] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.

[3] Martin Ward. Pigs from sausages? reengineering from assembler to c via fermat transformations. *Science of Computer Programming, Special Issue on Program Transformation 52*, pages 213–255, 2004.

[4] Martin Ward and K. H. Bennett. Formal methods for legacy systems. *Journal of Software Maintenance: Research and Practice*, 7(3):203–220, - 1995.

[5] Martin Ward and Tim Hardcastle. *WSL Programmer's Reference Manual*, Nov. 2003.

[6] Martin Ward and Hussein Zedan. Analysing and abstracting legacy assembler code via conditioned semantic slicing. 2006.

Web Links

[web1] The FermaT Engine
http://www.cse.dmu.ac.uk/~mward/fermat.html

[web2] The Software Technology Research Laboratory (DeMontfort University, Leicester)
http://www.cse.dmu.ac.uk/STRL/index.html

[web3] Software Migration Ltd.
http://www.smltd.com

[web4] Teach Yourself Scheme in Fixnum Days
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html